Уэс Маккинли

Python
и анализ данных

Python for Data Analysis

Wes McKinney

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Python и анализ данных

Уэс Маккинли

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2015

УДК 004.438Python:004.6
ББК 32.973.22
М15

М15 Уэс Маккинли

Python и анализ данных / Пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2015. – 482 с.: ил.

ISBN 978-5-97060-315-4

Книгу можно рассматривать как современное практическое введение в разработку научных приложений на Python, ориентированных на обработку данных. Описаны те части языка Python и библиотеки для него, которые необходимы для эффективного решения широкого круга аналитических задач: интерактивная оболочка IPython, библиотеки NumPy и pandas, библиотека для визуализации данных matplotlib и др.

Издание идеально подойдет как аналитикам, только начинающим осваивать обработку данных, так и опытным программистам на Python, еще не знакомым с научными приложениями.

УДК 004.438Python:004.6
ББК 32.973.22

Original English language edition published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472. Copyright © 2013 O'Reilly Media, Inc. Russian-language edition copyright © 2015 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-449-31979-3 (англ.)
ISBN 978-5-97060-315-4 (рус.)

Copyright © 2013 Wes McKinney.
© Оформление, перевод на русский язык, издание, ДМК Пресс, 2015

ОГЛАВЛЕНИЕ

Предисловие 12
Графические выделения 12

Глава 1. Предварительные сведения 13
О чем эта книга? 13
Почему именно Python? 14
Python как клей 14
Решение проблемы «двух языков» 15
Недостатки Python 15
Необходимые библиотеки для Python 16
NumPy 16
pandas 16
matplotlib 17
IPython 17
SciPy 18
Установка и настройка 18
Windows 19
Apple OS X 21
GNU/Linux 22
Python 2 и Python 3 23
Интегрированные среды разработки (IDE) 24
Сообщество и конференции 24
Структура книги 25
Примеры кода 25
Данные для примеров 25
Соглашения об импорте 25
Жаргон 26
Благодарности 26

Глава 2. Первые примеры 28
Набор данных 1.usa.gov с сайта bit.ly 28
Подсчет часовых поясов на чистом Python 30
Подсчет часовых поясов с помощью pandas 32
Набор данных MovieLens 1M 38
Измерение несогласия в оценках 42
Имена, которые давали детям в США за период с 1880 по 2010 год 43
Анализ тенденций в выборе имен 48
Выводы и перспективы 56

Глава 3. IPython: интерактивные вычисления и среда разработки 57
Основы IPython 58
Завершение по нажатию клавиши Tab 59
Интроспекция 60
Команда %run 61
Исполнение кода из буфера обмена 63
Комбинации клавиш 64
Исключения и обратная трассировка 65
Магические команды 66
Графическая консоль на базе Qt 68
Интеграция с matplotlib и режим pylab 68
История команд 70
Поиск в истории команд и повторное выполнение 70
Входные и выходные переменные 71
Протоколирование ввода-вывода 72
Взаимодействие с операционной системой 73
Команды оболочки и псевдонимы 73
Система закладок на каталоги 75
Средства разработки программ 75
Интерактивный отладчик 75
Хронометраж программы: %time и %timeit 80
Простейшее профилирование: %prun и %run -p 82
Построчное профилирование функции 83
HTML-блокнот в IPython 86
Советы по продуктивной разработке кода с использованием IPython ... 86
Перезагрузка зависимостей модуля 87
Советы по проектированию программ 88
Дополнительные возможности IPython 90
Делайте классы дружественными к IPython 90
Профили и конфигурирование 90
Благодарности 92

Глава 4. Основы NumPy: массивы и векторные вычисления ... 93
NumPy ndarray: объект многомерного массива 94
Создание ndarray 95
Тип данных для ndarray 97
Операции между массивами и скалярами 100
Индексирование и вырезание 100
Булево индексирование 104
Прихотливое индексирование 107
Транспонирование массивов и перестановка осей 108
Универсальные функции: быстрые поэлементные операции над массивами 109
Обработка данных с применением массивов 112
Запись логических условий в виде операций с массивами 113

Математические и статистические операции 115
Методы булевых массивов 116
Сортировка 117
Устранение дубликатов и другие теоретико-множественные операции....118
Файловый ввод-вывод массивов 119
Хранение массивов на диске в двоичном формате....119
Сохранение и загрузка текстовых файлов 120
Линейная алгебра.... 121
Генерация случайных чисел 122
Пример: случайное блуждание....123
Моделирование сразу нескольких случайных блужданий 125

Глава 5. Первое знакомство с pandas 127

Введение в структуры данных pandas 128
Объект Series 128
Объект DataFrame 131
Индексные объекты....137
Базовая функциональность.... 139
Переиндексация 139
Удаление элементов из оси....142
Доступ по индексу, выборка и фильтрация 143
Арифметические операции и выравнивание данных 146
Применение функций и отображение....150
Сортировка и ранжирование 151
Индексы по осям с повторяющимися значениями....154
Редукция и вычисление описательных статистик 155
Корреляция и ковариация 158
Уникальные значения, счетчики значений и членство....160
Обработка отсутствующих данных 162
Фильтрация отсутствующих данных 163
Иерархическое индексирование.... 166
Уровни переупорядочения и сортировки....169
Сводная статистика по уровню....170
Работа со столбцами DataFrame....170
Другие возможности pandas.... 172
Доступ по целочисленному индексу 172
Структура данных Panel....173

Глава 6. Чтение и запись данных, форматы файлов 175

Чтение и запись данных в текстовом формате 175
Чтение текстовых файлов порциями 181
Вывод данных в текстовом формате....182
Ручная обработка данных в формате с разделителями....184
Данные в формате JSON 186
XML и HTML: разбор веб-страниц....188
Разбор XML с помощью lxml.objectify....190
Двоичные форматы данных....192

Взаимодействие с HTML и Web API 194
Взаимодействие с базами данных 196
Чтение и сохранение данных в MongoDB 198

Глава 7. Переформатирование данных: очистка, преобразование, слияние, изменение формы 199

Комбинирование и слияние наборов данных 199
Слияние объектов DataFrame как в базах данных 200
Слияние по индексу 204
Конкатенация вдоль оси 207
Комбинирование перекрывающихся данных 211
Изменение формы и поворот 212
Изменение формы с помощью иерархического индексирования 213
Поворот из «длинного» в «широкий» формат 215
Преобразование данных 217
Устранение дубликатов 217
Преобразование данных с помощью функции или отображения 218
Замена значений 220
Переименование индексов осей 221
Дискретизация и раскладывание 222
Обнаружение и фильтрация выбросов 224
Перестановки и случайная выборка 226
Вычисление индикаторных переменных 227
Манипуляции со строками 229
Методы строковых объектов 230
Регулярные выражения 232
Векторные строковые функции в pandas 235
Пример: база данных о продуктах питания министерства сельского хозяйства США 237

Глава 8. Построение графиков и визуализация 244

Краткое введение в API библиотеки matplotlib 245
Рисунки и подграфики 246
Цвета, маркеры и стили линий 249
Риски, метки и надписи 251
Аннотации и рисование в подграфике 254
Сохранение графиков в файле 256
Конфигурирование matplotlib 257
Функции построения графиков в pandas 258
Линейные графики 258
Столбчатые диаграммы 260
Гистограммы и графики плотности 264
Диаграммы рассеяния 266
Нанесение данных на карту: визуализация данных о землетрясении на Гаити 267
Инструментальная экосистема визуализации для Python 273

Chaco 274
mayavi 274
Прочие пакеты 275
Будущее средств визуализации 275

Глава 9. Агрегирование данных и групповые операции 276

Механизм GroupBy 277
Обход групп 280
Выборка столбца или подмножества столбцов 281
Группировка с помощью словарей и объектов Series 282
Группировка с помощью функций 284
Группировка по уровням индекса 284

Агрегирование данных 285
Применение функций, зависящих от столбца, и нескольких функций 287
Возврат агрегированных данных в «неиндексированном» виде 289

Групповые операции и преобразования 290
Метод apply: часть общего принципа разделения–применения–объединения 292
Квантильный и интервальный анализ 294
Пример: подстановка зависящих от группы значений вместо отсутствующих 296
Пример: случайная выборка и перестановка 297
Пример: групповое взвешенное среднее и корреляция 299
Пример: групповая линейная регрессия 301

Сводные таблицы и кросс-табуляция 302
Таблицы сопряженности 304
Пример: база данных федеральной избирательной комиссии за 2012 год 305
Статистика пожертвований по роду занятий и месту работы 308
Распределение суммы пожертвований по интервалам 311
Статистика пожертвований по штатам 313

Глава 10. Временные ряды 316

Типы данных и инструменты, относящиеся к дате и времени 317
Преобразование между строкой и datetime 318

Основы работы с временными рядами 321
Индексирование, выборка, подмножества 322
Временные ряды с неуникальными индексами 324

Диапазоны дат, частоты и сдвиг 325
Генерация диапазонов дат 325
Частоты и смещения дат 326
Сдвиг данных (с опережением и с запаздыванием) 329

Часовые пояса 331
Локализация и преобразование 332
Операции над объектами Timestamp с учетом часового пояса 333
Операции между датами из разных часовых поясов 334
Периоды и арифметика периодов 335
Преобразование частоты периода 336

Квартальная частота периода 337
Преобразование временных меток в периоды и обратно 339
Создание PeriodIndex из массивов 340
Передискретизация и преобразование частоты 341
Понижающая передискретизация 342
Повышающая передискретизация и интерполяция 345
Передискретизация периодов 346
Графики временных рядов 348
Скользящие оконные функции 350
Экспоненциально взвешенные функции 353
Бинарные скользящие оконные функции 353
Скользящие оконные функции, определенные пользователем 355
Замечания о быстродействии и потреблении памяти 356

Глава 11. Финансовые и экономические приложения 358

О переформатировании данных 358
Временные ряды и выравнивание срезов 358
Операции над временными рядами с различной частотой 361
Время суток и выборка данных «по состоянию на» 364
Сращивание источников данных 366
Индексы доходности и кумулятивная доходность 368
Групповые преобразования и анализ 370
Оценка воздействия групповых факторов 372
Децильный и квартильный анализ 373
Другие примеры приложений 375
Стохастический граничный анализ 375
Роллинг фьючерсных контрактов 377
Скользящая корреляция и линейная регрессия 380

Глава 12. Дополнительные сведения о библиотеке NumPy ... 383

Иерархия типов данных в NumPy 384
Дополнительные манипуляции с массивами 385
Изменение формы массива 385
Упорядочение элементов массива в C и в Fortran 387
Конкатенация и разбиение массива 388
Повторение элементов: функции tile и repeat 390
Эквиваленты прихотливого индексирования: функции take и put 391
Укладывание 393
Укладывание по другим осям 394
Установка элементов массива с помощью укладывания 397
Дополнительные способы использования универсальных функций 398
Методы экземпляра u-функций 398
Пользовательские u-функции 400
Структурные массивы 401
Вложенные типы данных и многомерные поля 402
Зачем нужны структурные массивы? 403

Манипуляции со структурными массивами: numpy.lib.recfunctions 403
Еще о сортировке 403
Косвенная сортировка: методы argsort и lexsort 405
Альтернативные алгоритмы сортировки 406
Метод numpy.searchsorted: поиск элементов в отсортированном массиве 407
Класс matrix в NumPy 408
Дополнительные сведения о вводе-выводе массивов 410
Файлы, спроецированные на память 410
HDF5 и другие варианты хранения массива 412
Замечание о производительности 412
Важность непрерывной памяти 412
Другие возможности ускорения: Cython, f2py, C 414

Приложение. Основы языка Python 415
Интерпретатор Python 416
Основы 417
Семантика языка 417
Скалярные типы 425
Поток управления 431
Структуры данных и последовательности 437
Список 439
Встроенные функции последовательностей 443
Словарь 445
Множество 448
Списковое, словарное и множественное включение 450
Функции 452
Пространства имен, области видимости и локальные функции 453
Возврат нескольких значений 454
Функции являются объектами 455
Анонимные (лямбда) функции 456
Замыкания: функции, возвращающие функции 457
Расширенный синтаксис вызова с помощью *args и **kwargs 459
Каррирование: частичное фиксирование аргументов 459
Генераторы 460
Генераторные выражения 462
Модуль itertools 462
Файлы и операционная система 463

Предметный указатель 466

ПРЕДИСЛОВИЕ

За последние 10 лет вокруг языка Python образовалась и активно развивается целая экосистема библиотек с открытым исходным кодом. К началу 2011 года у меня сложилось стойкое ощущение, что нехватка централизованных источников учебных материалов по анализу данных и математической статистике становится камнем преткновения на пути молодых программистов на Python, которым такие приложения нужны по работе. Основные проекты, связанные с анализом данных (в особенности NumPy, IPython, matplotlib и pandas), к тому времени стали уже достаточно зрелыми, чтобы про них можно было написать книгу, которая не устареет сразу после выхода. Поэтому я набрался смелости заняться этим делом. Я был бы очень рад, если бы такая книга существовала в 2007 году, когда я приступал к использованию Python для анализа данных. Надеюсь, вам она окажется полезной, и вы сумеете с успехом воспользоваться описываемыми инструментами в собственной работе.

Графические выделения

В книге применяются следующие графические выделения:

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

`Моноширинный`

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

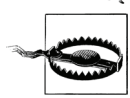
`Моноширинный полужирный`

Команды или иной текст, который должен быть введен пользователем буквально.

`Моноширинный курсив`

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.

Так обозначается совет, рекомендация или замечание общего характера.



Так обозначается предупреждение или предостережение.

ГЛАВА 1.
Предварительные сведения

О чем эта книга?

Эта книга посвящена вопросам преобразования, обработки, очистки данных и вычислениям на языке Python. Кроме того, она представляет собой современное практическое введение в научные и инженерные расчеты на Python, ориентированное на приложения для обработки больших объемов данных. Это книга о тех частях языка Python и написанных для него библиотек, которые необходимы для эффективного решения широкого круга задач анализа данных. Но в ней вы *не* найдете объяснений аналитических методов с привлечением Python в качестве языка реализации.

Говоря «данные», я имею в виду, прежде всего, *структурированные данные*; это намеренно расплывчатый термин, охватывающий различные часто встречающиеся виды данных, как то:

- многомерные списки (матрицы);
- табличные данные, когда данные в разных столбцах могут иметь разный тип (строки, числа, даты или еще что-то). Сюда относятся данные, которые обычно хранятся в реляционных базах или в файлах с запятой в качестве разделителя;
- данные, представленные в виде нескольких таблиц, связанных между собой по ключевым столбцам (то, что в SQL называется первичными и внешними ключами);
- равноотстоящие и неравноотстоящие временные ряды.

Этот список далеко не полный. Значительную часть наборов данных можно преобразовать к структурированному виду, более подходящему для анализа и моделирования, хотя сразу не всегда очевидно, как это сделать. В тех случаях, когда это не удается, иногда есть возможность извлечь из набора данных структурированное множество признаков. Например, подборку новостных статей можно преобразовать в таблицу частот слов, к которой затем применить анализ эмоциональной окраски.

Большинству пользователей электронных таблиц типа Microsoft Excel, пожалуй, самого широко распространенного средства анализа данных, такие виды данных хорошо знакомы.

Почему именно Python?

Для многих людей (и меня в том числе) Python — язык, в который нельзя не влюбиться. С момента своего появления в 1991 году Python стал одним из самых популярных динамических языков программирования наряду с Perl, Ruby и другими. Относительно недавно Python и Ruby приобрели особую популярность как средства создания веб-сайтов в многочисленных каркасах, например Rails (Ruby) и Django (Python). Такие языки часто называют *скриптовыми*, потому что они используются для быстрого написания небольших программ – *скриптов*. Лично мне термин «скриптовый язык» не нравится, потому что он наводит на мысль, будто для создания ответственного программного обеспечения язык не годится. Из всех интерпретируемых языков Python выделяется большим и активным сообществом *научных расчетов*. Применение Python для этой цели в промышленных и академических кругах значительно расширилось с начала 2000-х годов.

В области анализа данных и интерактивных научно-исследовательских расчетов с визуализацией результатов Python неизбежно приходится сравнивать со многими предметно-ориентированными языками программирования и инструментами – с открытым исходным кодом и коммерческими – такими, как R, MATLAB, SAS, Stata и другими. Сравнительно недавнее появление улучшенных библиотек для Python (прежде всего, pandas) сделало его серьезным конкурентом в решении задач манипулирования данными. В сочетании с достоинствами Python как универсального языка программирования это делает его отличным выбором для создания приложений обработки данных.

Python как клей

Своим успехом в качестве платформы для научных расчетов Python отчасти обязан простоте интеграции с кодом на C, C++ и FORTRAN. Во многих современных вычислительных средах применяется общий набор унаследованных библиотек, написанных на FORTRAN и C, содержащих реализации алгоритмов линейной алгебры, оптимизации, интегрирования, быстрого преобразования Фурье и других. Поэтому многочисленные компании и национальные лаборатории используют Python как «клей» для объединения написанных за 30 лет программ.

Многие программы содержат небольшие участки кода, на выполнение которых уходит большая часть времени, и большие куски «склеивающего кода», который выполняется нечасто. Во многих случаях время выполнения склеивающего кода несущественно, реальную отдачу дает оптимизация узких мест, которые иногда имеет смысл переписать на низкоуровневом языке типа C.

За последние несколько лет на одно из первых мест в области создания быстрых компилируемых расширений Python и организации интерфейса с кодом на C и C++ вышел проект Cython (*http://cython.org*).

Решение проблемы «двух языков»

Во многих организациях принято для научных исследований, создания опытных образцов и проверки новых идей использовать предметно-ориентированные языки типа MATLAB или R, а затем переносить удачные разработки в производственную систему, написанную на Java, C# или C++. Но все чаще люди приходят к выводу, что Python подходит не только для стадий исследования и создания прототипа, но и для построения самих производственных систем. Я полагаю, что компании все чаще будут выбирать этот путь, потому что использование одного и того же набора программных средств учеными и технологами несет несомненные выгоды организации.

Недостатки Python

Python – великолепная среда для создания приложений для научных расчетов и большинства систем общего назначения, но тем не менее существуют задачи, для которых Python не очень подходит.

Поскольку Python – интерпретируемый язык программирования, в общем случае написанный на нем код работает значительно медленнее, чем эквивалентный код на компилируемом языке типа Java или C++. Но поскольку *время программиста* обычно стоит гораздо дороже *времени процессора*, многих такой компромисс устраивает. Однако в приложениях, где задержка должна быть очень мала (например, в торговых системах с большим количеством транзакций), время, потраченное на программирование на низкоуровневом и не обеспечивающем максимальную продуктивность языке типа C++, во имя достижения максимальной производительности, будет потрачено не зря.

Python – не идеальный язык для программирования многопоточных приложений с высокой степенью параллелизма, особенно при наличии многих потоков, активно использующих процессор. Проблема связана с наличием *глобальной блокировки интерпретатора* (GIL) – механизма, который не дает интерпретатору исполнять более одной команды байт-кода Python в каждый момент времени. Объяснение технических причин существования GIL выходит за рамки этой книги, но на данный момент представляется, что GIL вряд ли скоро исчезнет. И хотя во многих приложениях обработки больших объектов данных для обеспечения приемлемого времени приходится организовывать кластер машин, встречаются все же ситуации, когда более желательна однопроцессная многопоточная система.

Я не хочу сказать, что Python вообще непригоден для исполнения многопоточного параллельного кода; просто такой код нельзя выполнять в одном процессе Python. Например, в проекте Cython реализована простая интеграция с OpenMP, написанной на C библиотеке параллельных вычислений, позволяющая распараллеливать циклы и тем самым значительно ускорять работу численных алгоритмов.

Необходимые библиотеки для Python

Для читателей, плохо знакомых с экосистемой Python и используемыми в книге библиотеками, я приведу краткий обзор библиотек.

NumPy

NumPy, сокращение от «Numerical Python», – основной пакет для выполнения научных расчетов на Python. Большая часть этой книги базируется на NumPy и построенных поверх него библиотек. В числе прочего он предоставляет:

- быстрый и эффективный объект многомерного массива *ndarray*;
- функции для выполнения вычислений над элементами одного массива или математических операций с несколькими массивами;
- средства для чтения и записи на диски наборов данных, представленных в виде массивов;
- операции линейной алгебры, преобразование Фурье и генератор случайных чисел;
- средства для интеграции с кодом, написанным на C, C++ или Fortran.

Помимо быстрых средств работы с массивами, одной из основных целей NumPy в части анализа данных является организация контейнера для передачи данных между алгоритмами. Как средство хранения и манипуляции данными массивы NumPy куда эффективнее встроенных в Python структур данных. Кроме того, библиотеки, написанные на низкоуровневом языке типа C или Fortran, могут работать с данными, хранящимися в массиве NumPy, вообще без копирования.

pandas

Библиотека pandas предоставляет структуры данных и функции, призванные сделать работу со структурированными данными простым, быстрым и выразительным делом. Как вы вскоре убедитесь, это один из основных компонентов, превращающих Python в мощный инструмент продуктивного анализа данных. Основной объект pandas, используемый в этой книге, – `DataFrame` – двумерная таблица, в которой строки и столбцы имеют метки:

```
>>> frame
     total_bill   tip     sex      smoker  day     time          size
1    16.99        1.01    Female   No      Sun     Dinner        2
2    10.34        1.66    Male     No      Sun     Dinner        3
3    21.01        3.5     Male     No      Sun     Dinner        3
4    23.68        3.31    Male     No      Sun     Dinner        2
5    24.59        3.61    Female   No      Sun     Dinner        4
6    25.29        4.71    Male     No      Sun     Dinner        4
7    8.77         2       Male     No      Sun     Dinner        2
8    26.88        3.12    Male     No      Sun     Dinner        4
9    15.04        1.96    Male     No      Sun     Dinner        2
10   14.78        3.23    Male     No      Sun     Dinner        2
```

Библиотека pandas сочетает высокую производительность средств работы с массивами, присущую NumPy, с гибкими возможностями манипулирования дан-

ными, свойственными электронным таблицам и реляционным базам данных (например, на основе SQL). Она предоставляет развитые средства индексирования, позволяющие без труда изменять форму наборов данных, формировать продольные и поперечные срезы, выполнять агрегирование и выбирать подмножества. В этой книге библиотека pandas будет нашим основным инструментом.

Для разработки финансовых приложений pandas предлагает богатый набор высокопроизводительных средств анализа временных рядов, специально ориентированных на финансовые данные. На самом деле, я изначально проектировал pandas как удобный инструмент анализа именно финансовых данных.

Пользователям языка статистических расчетов R название DataFrame покажется знакомым, потому что оно выбрано по аналогии с объектом `data.frame` в R. Однако они не идентичны: функциональность `data.frame` является собственным подмножеством той, что предлагает `DataFrame`. Хотя эта книга посвящена Python, я время от времени будут проводить сравнения с R, потому что это одна из самых распространенных сред анализа данных с открытым исходным кодом, знакомая многим читателям.

Само название pandas образовано как от *panel data* (панельные данные), применяемого в эконометрике термина для обозначения многомерных структурированных наборов данных, так и от фразы *Python data analysis*.

matplotlib

Библиотека matplotlib – самый популярный в Python инструмент для создания графиков и других способов визуализации двумерных данных. Первоначально она была написана Джоном Д. Хантером (John D. Hunter, JDH), а теперь сопровождается большой группой разработчиков. Она отлично подходит для создания графиков, пригодных для публикации. Интегрирована с IPython (см. ниже), что позволяет организовать удобное интерактивное окружение для визуализации и исследования данных. Графики *интерактивны* – можно увеличить масштаб какого-то участка графика и выполнять панорамирование с помощью панели инструментов в окне графика.

IPython

IPython – компонент стандартного набора инструментов научных расчетов на Python, который связывает все воедино. Он обеспечивает надежную высокопродуктивную среду для интерактивных и исследовательских расчетов. Это оболочка Python с дополнительными возможностями, имеющая целью ускорить написание, тестирование и отладку кода на Python. Особенно она полезна для работы с данными и их визуализации с помощью matplotlib. Я почти всегда использую IPython в собственной работе для прогона, отладки и тестирования кода.

Помимо стандартных средств консольной оболочки, IPython предоставляет:

- HTML-блокнот в духе программы Mathematica для подключения к IPython с помощью веб-браузера (подробнее об этом ниже);

- консоль с графическим интерфейсом пользователя на базе библиотеки Qt, включающую средства построения графиков, многострочный редактор и подсветку синтаксиса;
- инфраструктуру для интерактивных параллельных и распределенных вычислений.

Я посвятил отдельную главу оболочке IPython и способам оптимальной работы с ней. Настоятельно рекомендую использовать ее при чтении этой книги.

SciPy

SciPy – собрание пакетов, предназначенных для решения различных стандартных вычислительных задач. Вот несколько из них:

- `scipy.integrate`: подпрограммы численного интегрирования и решения дифференциальных уравнений;
- `scipy.linalg`: подпрограммы линейной алгебры и разложения матриц, дополняющие те, что включены в `numpy.linalg`;
- `scipy.optimize`: алгоритмы оптимизации функций (нахождения минимумов) и поиска корней;
- `scipy.signal`: средства обработки сигналов;
- `scipy.sparse`: алгоритмы работы с разреженными матрицами и решения разреженных систем линейных уравнений;
- `scipy.special`: обертка вокруг SPECFUN, написанной на Fortran библиотеке, содержащей реализации многих стандартных математических функций, в том числе гамма-функции;
- `scipy.stats`: стандартные непрерывные и дискретные распределения вероятностей (функции плотности вероятности, формирования выборки, функции непрерывного распределения вероятности), различные статистические критерии и дополнительные описательные статистики;
- `scipy.weave`: средство для встраивания кода на C++ с целью ускорения вычислений с массивами.

Совместно NumPy и SciPy образуют достаточно полную замену значительной части системы MATLAB и многочисленных дополнений к ней.

Установка и настройка

Поскольку Python используется в самых разных приложениях, не существует единственно верной процедуры установки Python и необходимых дополнительных пакетов. У многих читателей, скорее всего, нет среды, подходящей для научных применений Python и проработки этой книги, поэтому я приведу подробные инструкции для разных операционных систем. Я рекомендую использовать следующие базовые дистрибутивы Python:

Enthought Python Distribution: ориентированный на научные применения дистрибутив от компании Enthought (*http://www.enthought.com*). Включает EPDFree – бесплатный дистрибутив (содержит NumPy, SciPy, matplotlib, Chaco и IPython), и

EPD Full — полный комплект, содержащий более 100 научных пакетов для разных предметных областей. EPD Full бесплатно поставляется для академического использования, а прочим пользователям предлагается платная годовая подписка.

Python(x,y) (*http://pythonxy.googlecode.com*): бесплатный ориентированный на научные применения дистрибутив для Windows.

В инструкциях ниже подразумевается EPDFree, но вы можете выбрать любой другой подход, все зависит от ваших потребностей. На момент написания этой книги EPD включает версию Python 2.7, хотя в будущем это может измениться. После установки на вашей машине появятся следующие готовые к импорту пакеты:

- базовые пакеты для научных расчетов: NumPy, SciPy, matplotlib и IPython (входят в EPDFree);
- зависимости для IPython Notebook: tornado и pyzmq (также входят в EPDFree);
- pandas (версии 0.8.2 или выше).

По ходу чтения книги вам могут понадобиться также следующие пакеты: statsmodels, PyTables, PyQt (или эквивалентный ему PySide), xlrd, lxml, basemap, pymongo и requests. Они используются в разных примерах. Устанавливать их необязательно, и я рекомендую не торопиться с этим до момента, когда они действительно понадобятся. Например, сборка PyQt или PyTables из исходных кодов в OS X или Linux – довольно муторное дело. А пока нам важно получить минимальную работоспособную среду: EPDFree и pandas.

Сведения обо всех Python-пакетах, ссылки на установщики двоичных версий и другую справочную информацию можно найти в указателе пакетов Python (Python Package Index – PyPI, *http://pypi.python.org*). Заодно это отличный источник для поиска новых Python-пакетов.

Во избежание путаницы я не стану обсуждать более сложные средства управления окружением такие, как pip и virtualenv. В Интернете можно найти немало отличных руководств по ним.

Некоторым пользователям могут быть интересны альтернативные реализации Python, например IronPython, Jython или PyPy. Но для работы с инструментами, представленными в этой книге, в настоящее время необходим стандартный написанный на C интерпретатор Python, известный под названием CPython.

Windows

Для начала установки в Windows скачайте установщик EPDFree с сайта *http://www.enthought.com*; это файл с именем `epd_free-7.3-1-winx86.msi`. Запустите его и согласитесь с предлагаемым по умолчанию установочным каталогом `C:\Python27`. Если в этом каталоге уже был установлен Python, то рекомендую

предварительно удалить его вручную (или с помощью средства «Установка и удаление программ»).

Затем нужно убедиться, что Python прописался в системной переменной PATH и что не возникло конфликтов с ранее установленными версиями Python. Откройте консоль (выберите из меню «Пуск» пункт «Выполнить...» и наберите `cmd.exe`). Попробуйте запустить интерпретатор Python, введя команду `python`. Должно появиться сообщение, в котором указана установленная версия EPDFree:

```
C:\Users\Wes>python
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 14:30:37) on win32
Type "credits", "demo" or "enthought" for more information.
>>>
```

Если в сообщении указана другая версия EPD или вообще ничего не запускается, то нужно привести в порядок переменные среды Windows. В Windows 7 начните вводить фразу «environment variables» в поле поиска программ и выберите раздел `Edit environment variables for your account`. В Windows XP нужно перейти в `Панель управления > Система > Дополнительно > Переменные среды`. В появляющемся окне найдите переменную `Path`. В ней должны присутствовать следующие два каталога, разделенные точкой с запятой:

```
C:\Python27;C:\Python27\Scripts
```

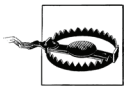
Если вы ранее устанавливали другие версии Python, удалите относящиеся в Python каталоги из системы и из переменных `Path`. После манипуляций с путями консоль необходимо перезапустить, чтобы изменения вступили в силу.

После того как Python удалось успешно запустить из консоли, необходимо установить pandas. Проще всего для этой цели загрузить подходящий двоичный установщик с сайта *http://pypi.python.org/pypi/pandas*. Для EPDFree это будет файл `pandas-0.9.0.win32-py2.7.exe`. После того как установщик отработает, запустим IPython и проверим, что все установилось правильно, для этого импортируем pandas и построим простой график с помощью matplotlib:

```
C:\Users\Wes>ipython --pylab
Python 2.7.3 |EPD_free 7.3-1 (32-bit)|
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??" for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

Если все нормально, то не будет никаких сообщений об ошибках и появится окно с графиком. Можно также проверить, что HTML-блокнот IPython HTML работает правильно:

```
$ ipython notebook --pylab=inline
```



Если в Windows вы обычно используете Internet Explorer, то для работы блокнота IPython, скорее всего, придется установить Mozilla Firefox или Google Chrome.

Дистрибутив EPDFree для Windows содержит только 32-разрядные исполняемые файлы. Если вам необходим 64-разрядный дистрибутив, то проще всего взять EPD Full. Если же вы предпочитаете устанавливать все с нуля и не платить EPD за подписку, то Кристоф Гольке (Christoph Gohlke) из Калифорнийского университета в Ирвайне опубликовал неофициальные двоичные 32- и 64-разрядные установщики для всех используемых в книге пакетов (*http://www.lfd.uci.edu/~gohlke/pythonlibs/*).

Apple OS X

Перед тем как приступать к установке в OS X, необходимо установить Xcode – комплект средств разработки ПО от Apple. Нам понадобится из него gcc – комплект компиляторов для C и C++. Установщик Xcode можно найти на установочном DVD, поставляемом вместе с компьютером, или скачать непосредственно с сайта Apple.

После установки Xcode запустите терминал (Terminal.app), перейдя в меню `Applications > Utilities`. Введите `gcc` и нажмите клавишу **Enter**. Должно появиться такое сообщение:

```
$ gcc
i686-apple-darwin10-gcc-4.2.1: no input files
```

Теперь необходимо установить EPDFree. Скачайте установщик, который должен представлять собой образ диска с именем вида `epd_free-7.3-1-macosx-i386.dmg`. Дважды щелкните мышью по файлу `dmg`-файлу, чтобы смонтировать его, а затем дважды щелкните по `mpkg`-файлу, чтобы запустить установщик.

Установщик автоматически добавит путь к исполняемому файлу EPDFree в ваш файл `.bash_profile`, его полный путь `/Users/ваше_имя/.bash_profile`:

```
# Setting PATH for EPD_free-7.3-1
PATH="/Library/Frameworks/Python.framework/Versions/Current/bin:${PATH}"
export PATH
```

Если на последующих шагах возникнут проблемы, проверьте файл `.bash_profile` – быть может, придется добавить указанный каталог в переменную PATH вручную.

Пришло время установить pandas. Выполните в терминале такую команду:

```
$ sudo easy_install pandas
Searching for pandas
Reading http://pypi.python.org/simple/pandas/
Reading http://pandas.pydata.org
Reading http://pandas.sourceforge.net
Best match: pandas 0.9.0
Downloading http://pypi.python.org/packages/source/p/pandas/pandas-0.9.0.zip
Processing pandas-0.9.0.zip
Writing /tmp/easy_install-H5mIX6/pandas-0.9.0/setup.cfg
Running pandas-0.9.0/setup.py -q bdist_egg --dist-dir /tmp/easy_install-H5mIX6/
pandas-0.9.0/egg-dist-tmp-RhLG0z

Adding pandas 0.9.0 to easy-install.pth file
Installed /Library/Frameworks/Python.framework/Versions/7.3/lib/python2.7/
site-packages/pandas-0.9.0-py2.7-macosx-10.5-i386.egg
Processing dependencies for pandas
Finished processing dependencies for pandas
```

Чтобы убедиться в работоспособности, запустите IPython в режиме Pylab и проверьте импорт pandas и интерактивное построение графика:

```
$ ipython --pylab
22:29 ~/VirtualBox VMs/WindowsXP $ ipython
Python 2.7.3 |EPD_free 7.3-1 (32-bit)| (default, Apr 12 2012, 11:28:34)
Type "copyright", "credits" or "license" for more information.

IPython 0.12.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

Welcome to pylab, a matplotlib-based Python environment [backend: WXAgg].
For more information, type 'help(pylab)'.

In [1]: import pandas

In [2]: plot(arange(10))
```

Если все нормально, появится окно графика, содержащее прямую линию.

GNU/Linux

В некоторые, но не во все дистрибутивы Linux включены достаточно актуальные версии всех необходимых Python-пакетов, и их можно установить с помощью встроенного средства управления пакетами, например `apt`. Я продемонстрирую установку на примере EPDFree, потому что она типична для разных дистрибутивов.

Детали варьируются в зависимости от дистрибутива Linux, я буду ориентироваться на дистрибутив Debian, на базе которого построены такие системы, как

Ubuntu и Mint. Установка в основных чертах производится так же, как для OS X, отличается только порядок установки EPDFree. Установщик представляет собой скрипт оболочки, запускаемый из терминала. В зависимости от разрядности системы нужно выбрать установщик типа `x86` (32-разрядный) или `x86_64` (64-разрядный). Имя соответствующего файла имеет вид `epd_free-7.3-1-rh5-x86_64.sh`. Для установки нужно выполнить такую команду:

```
$ bash epd_free-7.3-1-rh5-x86_64.sh
```

После подтверждения согласия с лицензией вам будет предложено указать место установки файлов EPDFree. Я рекомендую устанавливать их в свой домашний каталог, например `/home/wesm/epd` (вместо `wesm` подставьте свое имя пользователя).

После того как установщик отработает, добавьте в свою переменную `$PATH` подкаталог `bin` EPDFree. Если вы работаете с оболочкой bash (в Ubuntu она подразумевается по умолчанию), то нужно будет добавить такую строку в свой файл `.bashrc`:

```
export PATH=/home/wesm/epd/bin:$PATH
```

Естественно, вместо `/home/wesm/epd/` следует подставить свой установочный каталог. Затем запустите новый процесс терминала или повторно выполните файл `.bashrc` командой `source ~/.bashrc`.

Далее понадобится компилятор C, например gcc; во многие дистрибутивы Linux gcc включен, но это необязательно. В системах на базе Debian установка gcc производится командой:

```
sudo apt-get install gcc
```

Если набрать в командной строке слово `gcc`, то должно быть напечатано сообщение:

```
$ gcc
gcc: no input files
```

Теперь можно устанавливать pandas:

```
$ easy_install pandas
```

Если вы устанавливали EPDFree от имени пользователя root, то, возможно, придется добавить в начало команды слово `sudo` и ввести пароль. Проверка работоспособности производится так же, как в случае OS X.

Python 2 и Python 3

Сообщество Python в настоящее время переживает затянувшийся переход от семейства версий Python 2 к семейству Python 3. До появления Python 3.0 весь код на Python был обратно совместимым. Сообщество решило, что ради прогресса

языка необходимо внести некоторые изменения, которые лишат код этого свойства.

При написании этой книги я взял за основу Python 2.7, потому что большая часть научного сообщества Python еще не перешла на Python 3. Впрочем, если не считать немногих исключений, у вас не возникнет трудностей с исполнением приведенного в книге кода, даже если работать с Python 3.2.

Интегрированные среды разработки (IDE)

Когда меня спрашивают о том, какой средой разработки я пользуюсь, я почти всегда отвечаю: «IPython плюс текстовый редактор». Обычно я пишу программу и итеративно тестирую и отлаживаю ее по частям в IPython. Полезно также иметь возможность интерактивно экспериментировать с данными и визуально проверять, что в результате определенных манипуляций получается ожидаемый результат. Библиотеки pandas и NumPy спроектированы с учетом простоты использования в оболочке.

Но кто-то по-прежнему предпочитает работать в IDE, а не в текстовом редакторе. Интегрированные среды действительно предлагают много полезных «фенечек» типа автоматического завершения или быстрого доступа к документации по функциям и классам. Вот некоторые доступные варианты:

- Eclipse с подключаемым модулем PyDev;
- Python Tools для Visual Studio (для работающих в Windows);
- PyCharm;
- Spyder;
- Komodo IDE.

Сообщество и конференции

Помимо поиска в Интернете, существуют полезные списки рассылки, посвященные использованию Python в научных расчетах. Их участники быстро отвечают на вопросы. Вот некоторые из таких ресурсов:

- pydata: группа Google по вопросам, относящимся к использованию Python для анализа данных и pandas;
- pystatsmodels: вопросы, касающиеся statsmodels и pandas;
- numpy-discussion: вопросы, касающиеся NumPy;
- scipy-user: общие вопросы использования SciPy и Python для научных расчетов.

Я сознательно не публикую URL-адреса, потому что они часто меняются. Поиск в Интернете вам в помощь.

Ежегодно в разных странах проводят конференции для программистов на Python. PyCon и EuroPython – две самых крупных, проходящие соответственно в США и в Европе. SciPy и EuroSciPy – конференции, ориентированные на научные применения Python, где вы найдете немало «собратьев», если, прочитав эту книгу, захотите более плотно заняться анализом данных с помощью Python.

Структура книги

Если вы раньше никогда не программировали на Python, то имеет смысл начать с *конца* книги, где я поместил очень краткое руководство по синтаксису Python, языковым средствам и встроенным структурам данных: кортежам, спискам и словарям. Эти знания необходимы для чтения книги.

Книга начинается знакомством со средой IPython. Затем следует краткое введение в основные возможности NumPy, а более продвинутые рассматриваются в другой главе ближе к концу книги. Далее я знакомлю читателей с библиотекой pandas, а все остальные главы посвящены анализу данных с помощью pandas, NumPy и matplotlib (для визуализации). Я старался располагать материал последовательно, но иногда главы все же немного перекрываются.

Файлы с данными и материалы, относящиеся к каждой главе, размещаются в репозитории git на сайте GitHub:

```
http://github.com/pydata/pydata-book
```

Призываю вас скачать данные и пользоваться ими при воспроизведении примеров кода и экспериментах с описанными в книге инструментами. Я буду благодарен за любые добавления, скрипты, блокноты IPython и прочие материалы, которые вы захотите поместить в репозиторий книги для всеобщей пользы.

Примеры кода

Примеры кода в большинстве случаев показаны так, как выглядят в оболочке IPython: ввод и вывод.

```
In [5]: код
Out[5]: результат
```

Иногда для большей ясности несколько примеров кода показаны рядом. Их следует читать слева направо и исполнять по отдельности.

```
In [5]: код                 In [6]: код2
Out[5]: результат           Out[6]: результат2
```

Данные для примеров

Наборы данных для примеров из каждой главы находятся в репозитории на сайте GitHub: *http://github.com/pydata/pydata-book*. Вы можете получить их либо с помощью командной утилиты системы управления версиями git, либо скачав zip-файл репозитория с сайта.

Я стремился, чтобы в репозиторий попало все необходимое для воспроизведения примеров, но мог где-то ошибиться или что-то пропустить. В таком случае пишите мне на адрес `wesmckinn@gmail.com`.

Соглашения об импорте

В сообществе Python принят ряд соглашений об именовании наиболее употребительных модулей:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Это означает, что `np.arange` – ссылка на функцию `arange` в пакете NumPy. Так делается, потому что импорт всех имен из большого пакета, каким является NumPy (`from numpy import *`), считается среди разработчиков на Python дурным тоном.

Жаргон

Я употребляю некоторые термины, встречающиеся как в программировании, так и в науке о данных, с которыми вы, возможно, незнакомы. Поэтому приведу краткие определения.

- *Переформатирование (Munge/Munging/Wrangling)*
  Процесс приведения неструктурированных и (или) замусоренных данных к структурированной или чистой форме. Слово вошло в лексикон многих современных специалистов по анализу данных.
- *Псевдокод*
  Описание алгоритма или процесса в форме, напоминающей код, хотя фактически это не есть корректный исходный код на каком-то языке программирования.
- *Синтаксическая глазурь (syntactic sugar)*
  Синтаксическая конструкция, которая не добавляет какую-то новую функциональность, а лишь вносит дополнительное удобство или позволяет сделать код короче.

Благодарности

Я не смог бы написать эту книгу без помощи со стороны многих людей.

Из сотрудников издательства O'Reilly я крайне признателен своим редакторам Меган Бланшетт (Meghan Blanchette) и Джулии Стил (Julie Steele), которые направляли меня на протяжении всей работы. Майк Лукидес (Mike Loukides) работал со мной на этапе подготовки предложения и помог превратить замысел в реальность.

Техническое рецензирование осуществляла большая группа людей. В частности, Мартин Блез (Martin Blais) и Хью Уайт (Hugh White) оказали неоценимую помощь в подборе примеров, повышении ясности изложения и улучшении структуры книги в целом. Джеймс Лонг (James Long), Дрю Конвей (Drew Conway), Фернандо Перес (Fernando Pérez), Брайан Грейнджер (Brian Granger), Томас Клюйвер (Thomas Kluyver), Адам Клейн (Adam Klein), Джош Клейн (Josh Klein), Чань Ши (Chang She) и Стефан ван дер Вальт (Stéfan van der Walt) просмотрели по одной или по несколько глав под разными углами зрения.

Много идей по поводу примеров и наборов данных мне предложили друзья и коллеги по сообществу анализа данных, в том числе: Майк Дьюар (Mike Dewar),

Джефф Хаммербахер (Jeff Hammerbacher), Джеймс Джондроу (James Johndrow), Кристиан Лам (Kristian Lum), Адам Клейн, Хилари Мейсон (Hilary Mason), Чань Ши и Эшли Уильямс (Ashley Williams).

Разумеется, я в долгу перед многими лидерами сообщества «научного Python», которые создали открытый исходный код, легший в основу моих исследований, и оказывали мне поддержку на протяжении всей работы над книгой: группа разработки ядра IPython (Фернандо Перес, Брайан Грейнджер, Мин Рэган-Келли (Min Ragan-Kelly), Томас Клюйвер и другие), Джон Хантер (John Hunter), Скиппер Сиболд (Skipper Seabold), Трэвис Олифант (Travis Oliphant), Питер Уонг (Peter Wang), Эрик Джоунз (Eric Jones), Роберт Керн (Robert Kern), Джозеф Перктольд (Josef Perktold), Франческ Олтед (Francesc Alted), Крис Фоннесбек (Chris Fonnesbeck) и многие, многие другие, перечислять которых здесь нет никакой возможности. Меня также поддерживали, подбадривали и делились идеями Дрю Конвей, Шон Тэйлор (Sean Taylor), Джузеппе Палеолого (Giuseppe Paleologo), Джаред Ландер (Jared Lander), Дэвид Эпштейн (David Epstein), Джог Кровас (John Krowas), Джошуа Блум (Joshua Bloom), Дэн Пилсфорт (Den Pilsworth), Джон Майлз-Уайт (John Myles-White) и многие другие, имена которых я сейчас не могу вспомнить.

Я также благодарен многим, кто оказал влияние на мое становление как ученого. В первую очередь, это мои бывшие коллеги по компании AQR, которые поддерживали мою работу над pandas в течение многих лет: Алекс Рейфман (Alex Reyfman), Майкл Вонг (Michael Wong), Тим Сарджен (Tim Sargen), Октай Курбанов (Oktay Kurbanov), Мэтью Щанц (Matthew Tschantz), Рони Израэлов (Roni Israelov), Майкл Кац (Michael Katz), Крис Уга (Chris Uga), Прасад Раманан (Prasad Ramanan), Тэд Сквэр (Ted Square) и Хун Ким (Hoon Kim). И наконец, благодарю моих университетских наставников Хэйнса Миллера (МТИ) и Майка Уэста (университет Дьюк).

Если говорить о личной жизни, то я благодарен Кэйси Динкин (Casey Dinkin), чью каждодневную поддержку невозможно переоценить, ту, которая терпела перепады моего настроения, когда я пытался собрать окончательный вариант рукописи в дополнение к своему и так уже перегруженному графику. А также моим родителям, Биллу и Ким, которые учили меня никогда не отступать от мечты и не соглашаться на меньшее.

ГЛАВА 2.
Первые примеры

В этой книге рассказывается об инструментах, позволяющих продуктивно работать с данными в программах на языке Python. Хотя конкретные цели читателей могут быть различны, почти любую задачу можно отнести к одной из нескольких широких групп:

- *Взаимодействие с внешним миром*
  Чтение и запись данных, хранящихся в файлах различных форматов и в базах данных.
- *Подготовка*
  Очистка, переформатирование, комбинирование, нормализация, изменение формы, формирование продольных и поперечных срезов, преобразование данных для анализа.
- *Преобразование*
  Применение математических и статистических операций к группам наборов данных для порождения новых наборов. Например, агрегирование большой таблицы по групповым переменным.
- *Моделирование и расчет*
  Соединение данных со статистическими моделями, алгоритмами машинного обучения и другими вычислительными средствами.
- *Презентация*
  Создание интерактивных или статических графических представлений или текстовых сводок.

В этой главе я продемонстрирую несколько наборов данных и что с ними можно делать. Примеры преследуют только одну цель — возбудить у вас интерес, поэтому объяснения будут весьма общими. Не расстраивайтесь, если у вас пока нет опыта работа с описываемыми инструментами; они будут подробно рассматриваться на протяжении всей книги. В примерах кода вы встретите строки вида `In [15]:`, они взяты напрямую из оболочки IPython.

Набор данных 1.usa.gov с сайта bit.ly

В 2011 году служба сокращения URL-адресов bit.ly заключила партнерское соглашение с сайтом правительства США `usa.gov` о синхронном предоставлении

анонимных данных о пользователях, которые сокращают ссылки, заканчивающиеся на `.gov` или `.mil`. На момент написания этой книги помимо синхронной ленты, каждый час формируются мгновенные снимки, доступные в виде текстовых файлов[1].

В мгновенном снимке каждая строка представлена в формате JSON (JavaScript Object Notation), широко распространенном в веб. Например, первая строка файла выглядит примерно так:

```
In [15]: path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
In [16]: open(path).readline()
Out[16]: '{ "a": "Mozilla\\/5.0 (Windows NT 6.1; WOW64) AppleWebKit\\/535.11
(KHTML, like Gecko) Chrome\\/17.0.963.78 Safari\\/535.11", "c": "US", "nk": 1,
"tz": "America\\/New_York", "gr": "MA", "g": "A6qOVH", "h": "wfLQtf", "l":
"orofrog", "al": "en-US,en;q=0.8", "hh": "1.usa.gov", "r":
"http:\\/\\/www.facebook.com\\/l\\/7AQEFzjSi\\/1.usa.gov\\/wfLQtf", "u":
"http:\\/\\/www.ncbi.nlm.nih.gov\\/pubmed\\/22415991", "t": 1331923247, "hc":
1331822918, "cy": "Danvers", "ll": [ 42.576698, -70.954903 ] }\n'
```

Для Python имеется много встроенных и сторонних модулей, позволяющих преобразовать JSON-строку в объект словаря Python. Ниже я воспользовался модулем `json`; принадлежащая ему функция `loads` вызывается для каждой строки скачанного мной файла:

```
import json
path = 'ch02/usagov_bitly_data2012-03-16-1331923249.txt'
records = [json.loads(line) for line in open(path)]
```

Для тех, кто никогда не программировал на Python, скажу, что выражение в последней строке называется *списковым включением*; это краткий способ применить некую операцию (в данном случае `json.loads`) к коллекции строк или других объектов. Очень удобно – в случае, когда итерация применяется к описателю открытого файла, мы получаем последовательность прочитанных из него строк. Получившийся в результате объект `records` представляет собой список словарей Python:

```
In [18]: records[0]
Out[18]:
{u'a': u'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.11 (KHTML, like
Gecko) Chrome/17.0.963.78 Safari/535.11',
 u'al': u'en-US,en;q=0.8',
 u'c': u'US',
 u'cy': u'Danvers',
 u'g': u'A6qOVH',
 u'gr': u'MA',
 u'h': u'wfLQtf',
 u'hc': 1331822918,
 u'hh': u'1.usa.gov',
 u'l': u'orofrog',
 u'll': [42.576698, -70.954903],
```

[1] http://www.usa.gov/About/developer-resources/1usagov.shtml

```
 u'nk': 1,
 u'r': u'http://www.facebook.com/l/7AQEFzjSi/1.usa.gov/wfLQtf',
 u't': 1331923247,
 u'tz': u'America/New_York',
 u'u': u'http://www.ncbi.nlm.nih.gov/pubmed/22415991'}
```

Отметим, что в Python индексы начинаются с 0, а не с 1, как в некоторых других языках (например, R). Теперь нетрудно выделить интересующие значения из каждой записи, передав строку, содержащую ключ:

```
In [19]: records[0]['tz']
Out[19]: u'America/New_York'
```

Буква `u` перед знаком кавычки означает *unicode* – стандартную кодировку строк. Отметим, что IPython показывает *представление* объекта строки часового пояса, а не его печатный эквивалент:

```
In [20]: print records[0]['tz']
America/New_York
```

Подсчет часовых поясов на чистом Python

Допустим, что нас интересуют часовые пояса, чаще всего встречающиеся в наборе данных (поле `tz`). Решить эту задачу можно разными способами. Во-первых, можно извлечь список часовых поясов, снова воспользовавшись списковым включением:

```
In [25]: time_zones = [rec['tz'] for rec in records]
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/home/wesm/book_scripts/whetting/<ipython> in <module>()
----> 1 time_zones = [rec['tz'] for rec in records]
KeyError: 'tz'
```

Вот те раз! Оказывается, что не во всех записях есть поле часового пояса. Это легко поправить, добавив проверку `if 'tz' in rec` в конец спискового включения:

```
In [26]: time_zones = [rec['tz'] for rec in records if 'tz' in rec]
In [27]: time_zones[:10]
Out[27]:
[u'America/New_York',
 u'America/Denver',
 u'America/New_York',
 u'America/Sao_Paulo',
 u'America/New_York',
 u'America/New_York',
 u'Europe/Warsaw',
 u'',
 u'',
 u'']
```

Мы видим, что уже среди первых 10 часовых поясов встречаются неизвестные (пустые). Их можно было бы тоже отфильтровать, но я пока оставлю. Я покажу два способа подсчитать количество часовых поясов: трудный (в котором используется только стандартная библиотека Python) и легкий (с помощью pandas). Для подсчета можно завести словарь для хранения счетчиков и обойти весь список часовых поясов:

```
def get_counts(sequence):
 counts = {}
 for x in sequence:
  if x in counts:
   counts[x] += 1
  else:
   counts[x] = 1
  return counts
```

Зная стандартную библиотеку Python немного получше, можно было бы записать то же самое короче:

```
from collections import defaultdict

def get_counts2(sequence):
 counts = defaultdict(int) # values will initialize to 0
 for x in sequence:
  counts[x] += 1
 return counts
```

Чтобы можно было повторно воспользоваться этим кодом, я поместил его в функцию. Чтобы применить его к часовым поясам, достаточно передать этой функции список `time_zones`:

```
In [31]: counts = get_counts(time_zones)

In [32]: counts['America/New_York']
Out[32]: 1251

In [33]: len(time_zones)
Out[33]: 3440
```

Чтобы получить только первые 10 часовых поясов со счетчиками, придется поколдовать над словарем:

```
def top_counts(count_dict, n=10):
 value_key_pairs = [(count, tz) for tz, count in count_dict.items()]
 value_key_pairs.sort()
 return value_key_pairs[-n:]
```

В результате получим:

```
In [35]: top_counts(counts)
Out[35]:
[(33, u'America/Sao_Paulo'),
```

```
 (35, u'Europe/Madrid'),
 (36, u'Pacific/Honolulu'),
 (37, u'Asia/Tokyo'),
 (74, u'Europe/London'),
 (191, u'America/Denver'),
 (382, u'America/Los_Angeles'),
 (400, u'America/Chicago'),
 (521, u''),
 (1251, u'America/New_York')]
```

Пошарив в стандартной библиотеке Python, можно найти класс `collections.Counter`, который позволяет решить задачу гораздо проще:

```
In [49]: from collections import Counter

In [50]: counts = Counter(time_zones)

In [51]: counts.most_common(10)
Out[51]:
[(u'America/New_York', 1251),
 (u'', 521),
 (u'America/Chicago', 400),
 (u'America/Los_Angeles', 382),
 (u'America/Denver', 191),
 (u'Europe/London', 74),
 (u'Asia/Tokyo', 37),
 (u'Pacific/Honolulu', 36),
 (u'Europe/Madrid', 35),
 (u'America/Sao_Paulo', 33)]
```

Подсчет часовых поясов с помощью pandas

Основной в библиотеке pandas является структура данных *DataFrame*, которую можно представлять себе как таблицу. Создать экземпляр DataFrame из исходного набора записей просто:

```
In [289]: from pandas import DataFrame, Series

In [290]: import pandas as pd

In [291]: frame = DataFrame(records)

In [292]: frame

Out[292]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3560 entries, 0 to 3559
Data columns:
_heartbeat_   120 non-null values
a            3440 non-null values
al           3094 non-null values
c            2919 non-null values
cy           2919 non-null values
g            3440 non-null values
```

```
gr           2919 non-null values
h            3440 non-null values
hc           3440 non-null values
hh           3440 non-null values
kw             93 non-null values
l            3440 non-null values
ll           2919 non-null values
nk           3440 non-null values
r            3440 non-null values
t            3440 non-null values
tz           3440 non-null values
u            3440 non-null values
dtypes: float64(4), object(14)

In [293]: frame['tz'][:10]
Out[293]:
0     America/New_York
1       America/Denver
2     America/New_York
3    America/Sao_Paulo
4     America/New_York
5     America/New_York
6        Europe/Warsaw
7
8
9
Name: tz
```

На выходе по запросу `frame` мы видим сводное представление, которое показывается для больших объектов DataFrame. Объект Series, возвращаемый в ответ на запрос `frame['tz']`, имеет метод `value_counts`, который дает как раз то, что нам нужно:

```
In [294]: tz_counts = frame['tz'].value_counts()

In [295]: tz_counts[:10]
Out[295]:

America/New_York    1251
                     521
America/Chicago     400
America/Los_Angeles 382
America/Denver      191
Europe/London        74
Asia/Tokyo           37
Pacific/Honolulu     36
Europe/Madrid        35
America/Sao_Paulo    33
```

После этого можно с помощью библиотеки matplotlib построить график этих данных. Возможно, придется слегка подправить их, подставив какое-нибудь значение вместо неизвестных и отсутствующих часовых поясов. Заменить отсутствующие (NA) значения позволяет функция `fillna`, а неизвестные значения (пустые строки) можно заменить с помощью булевой индексации массива:

```
In [296]: clean_tz = frame['tz'].fillna('Missing')

In [297]: clean_tz[clean_tz == ''] = 'Unknown'

In [298]: tz_counts = clean_tz.value_counts()

In [299]: tz_counts[:10]
Out[299]:
America/New_York    1251
Unknown              521
America/Chicago      400
America/Los_Angeles  382
America/Denver       191
Missing              120
Europe/London         74
Asia/Tokyo            37
Pacific/Honolulu      36
Europe/Madrid         35
```

Для построения горизонтальной столбчатой диаграммы можно применить метод `plot` к объектам `counts`:

```
In [301]: tz_counts[:10].plot(kind='barh', rot=0)
```

Результат показан на рис. 2.1. Ниже мы рассмотрим и другие инструменты для работы с такими данными. Например, поле `a` содержит информацию о браузере, устройстве или приложении, выполнившем сокращение URL:

```
In [302]: frame['a'][1]
Out[302]: u'GoogleMaps/RochesterNY'

In [303]: frame['a'][50]
Out[303]: u'Mozilla/5.0 (Windows NT 5.1; rv:10.0.2) Gecko/20100101 Firefox/10.0.2'

In [304]: frame['a'][51]
Out[304]: u'Mozilla/5.0 (Linux; U; Android 2.2.2; en-us; LG-P925/V10e Build/
FRG83G) AppleWebKit/533.1 (KHTML, like Gecko) Version/4.0 Mobile Safari/533
```

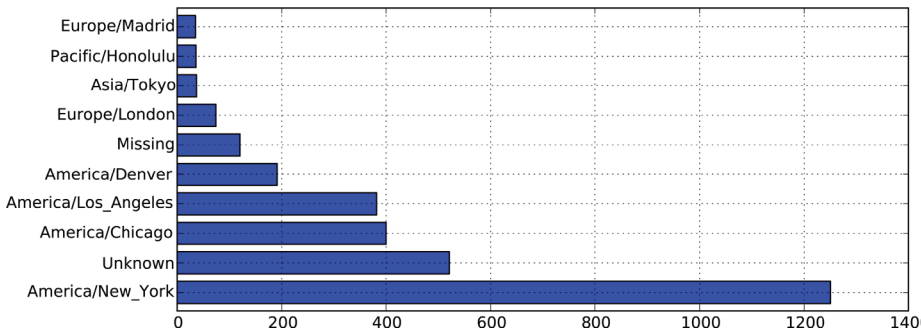


Рис. 2.1. Первые 10 часовых поясов из набора данных 1.usa.gov

Выделение всей интересной информации из таких строк «пользовательских агентов» поначалу может показаться пугающей задачей. По счастью, на деле все не так плохо – нужно только освоить встроенные в Python средства для работы со строками и регулярными выражениями. Например, вот как можно вырезать из строки первую лексему (грубо описывающую возможности браузера) и представить поведение пользователя в другом разрезе:

```
In [305]: results = Series([x.split()[0] for x in frame.a.dropna()])

In [306]: results[:5]
Out[306]:
0    Mozilla/5.0
1    GoogleMaps/RochesterNY
2    Mozilla/4.0
3    Mozilla/5.0
4    Mozilla/5.0

In [307]: results.value_counts()[:8]
Out[307]:
Mozilla/5.0                 2594
Mozilla/4.0                  601
GoogleMaps/RochesterNY       121
Opera/9.80                    34
TEST_INTERNET_AGENT           24
GoogleProducer                21
Mozilla/6.0                    5
BlackBerry8520/5.0.0.681       4
```

Предположим теперь, что требуется разделить пользователей в первых 10 часовых поясах на работающих в Windows и всех прочих. Упростим задачу, предположив, что пользователь работает в Windows, если строка агента содержит подстроку `'Windows'`. Но строка агента не всегда присутствует, поэтому записи, в которых ее нет, я исключу:

```
In [308]: cframe = frame[frame.a.notnull()]
```

Мы хотим вычислить значение, показывающее, относится строка к пользователю Windows или нет:

```
In [309]: operating_system = np.where(cframe['a'].str.contains('Windows'),
  .....:                              'Windows', 'Not Windows')

In [310]: operating_system[:5]
Out[310]:
0        Windows
1    Not Windows
2        Windows
3    Not Windows
4        Windows
Name: a
```

Затем мы можем сгруппировать данные по часовому поясу и только что сформированному столбцу с типом операционной системы:

```
In [311]: by_tz_os = cframe.groupby(['tz', operating_system])
```

Групповые счетчики по аналогии с рассмотренной выше функцией `value_counts` можно вычислить с помощью функции `size`. А затем преобразовать результат в таблицу с помощью `unstack`:

```
In [312]: agg_counts = by_tz_os.size().unstack().fillna(0)

In [313]: agg_counts[:10]
Out[313]:
a                      Not Windows    Windows
tz
                                245        276
Africa/Cairo                      0          3
Africa/Casablanca                 0          1
Africa/Ceuta                      0          2
Africa/Johannesburg               0          1
Africa/Lusaka                     0          1
America/Anchorage                 4          1
America/Argentina/Buenos_Aires    1          0
America/Argentina/Cordoba         0          1
America/Argentina/Mendoza         0          1
```

Наконец, выберем из полученной таблицы первые 10 часовых поясов. Для этого я построю массив косвенных индексов `agg_counts` по счетчикам строк:

```
# Нужен для сортировки в порядке возрастания
In [314]: indexer = agg_counts.sum(1).argsort()

In [315]: indexer[:10]
Out[315]:
tz
                                  24
Africa/Cairo                      20
Africa/Casablanca                 21
Africa/Ceuta                      92
Africa/Johannesburg               87
Africa/Lusaka                     53
America/Anchorage                 54
America/Argentina/Buenos_Aires    57
America/Argentina/Cordoba         26
America/Argentina/Mendoza         55
```

А затем с помощью `take` расположу строки в порядке, определяемом этим индексом, и оставлю только последние 10:

```
In [316]: count_subset = agg_counts.take(indexer)[-10:]

In [317]: count_subset
Out[317]:
```

```
a                   Not Windows       Windows
tz
America/Sao_Paulo             13            20
Europe/Madrid                 16            19
Pacific/Honolulu               0            36
Asia/Tokyo                     2            35
Europe/London                 43            31
America/Denver               132            59
America/Los_Angeles          130           252
America/Chicago              115           285
                             245           276
America/New_York             339           912
```

Теперь можно построить столбчатую диаграмму, как и в предыдущем примере. Только на этот раз я сделаю ее штабельной, передав параметр `stacked=True` (см. рис. 2.2):

```
In [319]: count_subset.plot(kind='barh', stacked=True)
```

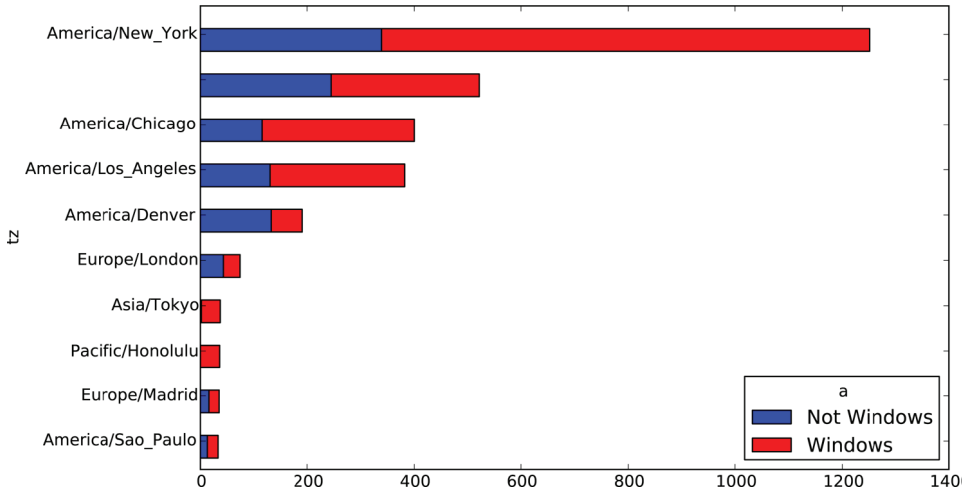


Рис. 2.2. Первые 10 часовых поясов с выделением пользователей Windows и прочих

Из этой диаграммы трудно наглядно представить, какова процентная доля пользователей Windows в каждой группе, но строки легко можно нормировать, так чтобы в сумме получилась 1, а затем построить диаграмму еще раз (рис. 2.3):

```
In [321]: normed_subset = count_subset.div(count_subset.sum(1), axis=0)
In [322]: normed_subset.plot(kind='barh', stacked=True)
```

Все использованные нами методы будут подробно рассмотрены в последующих главах.

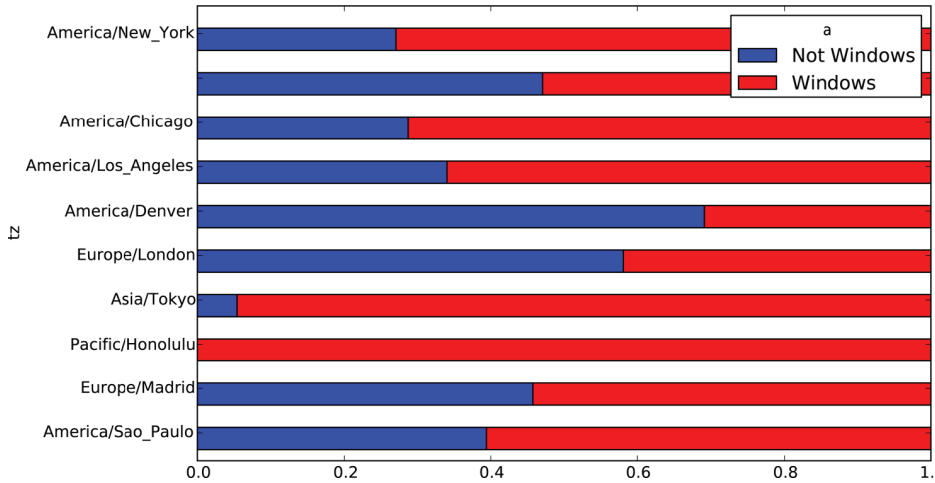


Рис. 2.3. Процентная доля пользователей Windows и прочих в первых 10 часовых поясах

Набор данных MovieLens 1M

Исследовательская группа GroupLens Research (*http://www.grouplens.org/node/73*) предлагает несколько наборов данных о рейтингах фильмов, проставленных пользователями сайта MovieLens в конце 1990-х – начале 2000-х. Наборы содержат рейтинги фильмов, метаданные о фильмах (жанр и год выхода) и демографические данные о пользователях (возраст, почтовый индекс, пол и род занятий). Такие данные часто представляют интерес для разработки систем рекомендования, основанных на алгоритмах машинного обучения. И хотя в этой книге методы машинного обучения не рассматриваются, я все же покажу, как формировать продольные и поперечные срезы таких наборов данных с целью привести их к нужному виду.

Набор MovieLens 1M содержит 1 миллион рейтингов 4000 фильмов, проставленных 6000 пользователей. Данные распределены по трем таблицам: рейтинги, информация о пользователях и информация о фильмах. После распаковки zip-файла каждую таблицу можно загрузить в отдельный объект DataFrame с помощью метода `pandas.read_table`:

```
import pandas as pd

unames = ['user_id', 'gender', 'age', 'occupation', 'zip']
users = pd.read_table('ml-1m/users.dat', sep='::', header=None,
                      names=unames)

rnames = ['user_id', 'movie_id', 'rating', 'timestamp']
ratings = pd.read_table('ml-1m/ratings.dat', sep='::', header=None,
                        names=rnames)

mnames = ['movie_id', 'title', 'genres']
```

```
movies = pd.read_table('ml-1m/movies.dat', sep='::', header=None,
                       names=mnames)
```

Проверить, все ли прошло удачно, можно, посмотрев на первые несколько строк каждого DataFrame с помощью встроенного в Python синтаксиса вырезания:

```
In [334]: users[:5]
Out[334]:
   user_id gender  age  occupation    zip
0        1      F    1          10  48067
1        2      M   56          16  70072
2        3      M   25          15  55117
3        4      M   45           7  02460
4        5      M   25          20  55455

In [335]: ratings[:5]
Out[335]:
   user_id  movie_id  rating  timestamp
0        1      1193       5  978300760
1        1       661       3  978302109
2        1       914       3  978301968
3        1      3408       4  978300275
4        1      2355       5  978824291

In [336]: movies[:5]
Out[336]:
  movie_id                               title                        genres
0        1                    Toy Story (1995)   Animation|Children's|Comedy
1        2                      Jumanji (1995)  Adventure|Children's|Fantasy
2        3             Grumpier Old Men (1995)                Comedy|Romance
3        4            Waiting to Exhale (1995)                  Comedy|Drama
4        5  Father of the Bride Part II (1995)                        Comedy

In [337]: ratings
Out[337]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id    1000209   non-null values
movie_id   1000209   non-null values
rating     1000209   non-null values
timestamp  1000209   non-null values
dtypes: int64(4)
```

Отметим, что возраст и род занятий кодируются целыми числами, а расшифровка приведена в прилагаемом к набору данных файлу `README`. Анализ данных, хранящихся в трех таблицах, – непростая задача. Пусть, например, требуется вычислить средние рейтинги для конкретного фильма в разрезе пола и возраста. Как мы увидим, это гораздо легче сделать, если предварительно объединить все данные в одну таблицу. Применяя функцию `merge` из библиотеки pandas, мы сначала объединим `ratings` с `users`, а затем результат объединим с `movies`. Pandas определяем, по каким столбцам объединять (или *соединять*), ориентируясь на совпадение имен:

```
In [338]: data = pd.merge(pd.merge(ratings, users), movies)
In [339]: data

Out[339]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000209 entries, 0 to 1000208
Data columns:
user_id    1000209  non-null values
movie_id   1000209  non-null values
rating     1000209  non-null values
timestamp  1000209  non-null values
gender     1000209  non-null values
age        1000209  non-null values
occupation 1000209  non-null values
zip        1000209  non-null values
title      1000209  non-null values
genres     1000209  non-null values
dtypes: int64(6), object(4)

In [340]: data.ix[0]
Out[340]:
user_id              1
movie_id             1
rating               5
timestamp    978824268
gender               F
age                  1
occupation          10
zip              48067
title       Toy Story (1995)
genres      Animation|Children's|Comedy
Name: 0
```

В таком виде агрегирование рейтингов, сгруппированных по одному или нескольким атрибутам пользователя или фильма, для человека, хоть немного знакомого с pandas, не представляет никаких трудностей. Чтобы получить средние рейтинги фильмов при группировке по полу, воспользуемся методом `pivot_table`:

```
In [341]: mean_ratings = data.pivot_table('rating', rows='title',
   .....:                                  cols='gender', aggfunc='mean')

In [342]: mean_ratings[:5]
Out[342]:
gender                                F         M
title
$1,000,000 Duck (1971)         3.375000  2.761905
'Night Mother (1986)           3.388889  3.352941
'Til There Was You (1997)      2.675676  2.733333
'burbs, The (1989)             2.793478  2.962085
...And Justice for All (1979)  3.828571  3.689024
```

В результате получается еще один объект DataFrame, содержащий средние рейтинги, в котором метками строк являются общее количество оценок фильма, а

метками столбцов – обозначения полов. Сначала я оставлю только фильмы, получившие не менее 250 оценок (число выбрано совершенно произвольно); для этого сгруппирую данные по названию и с помощью метода `size()` получу объект Series, содержащий размеры групп для каждого наименования:

```
In [343]: ratings_by_title = data.groupby('title').size()

In [344]: ratings_by_title[:10]
Out[344]:
title
$1,000,000 Duck (1971)              37
'Night Mother (1986)                70
'Til There Was You (1997)           52
'burbs, The (1989)                 303
...And Justice for All (1979)      199
1-900 (1994)                         2
10 Things I Hate About You (1999)  700
101 Dalmatians (1961)              565
101 Dalmatians (1996)              364
12 Angry Men (1957)                616

In [345]: active_titles = ratings_by_title.index[ratings_by_title >= 250]

In [346]: active_titles
Out[346]:
Index(['burbs, The (1989), 10 Things I Hate About You (1999),
       101 Dalmatians (1961), ..., Young Sherlock Holmes (1985),
       Zero Effect (1998), eXistenZ (1999)], dtype=object)
```

Затем для отбора строк из приведенного выше объекта `mean_ratings` воспользуемся индексом фильмов, получивших не менее 250 оценок:

```
In [347]: mean_ratings = mean_ratings.ix[active_titles]

In [348]: mean_ratings
Out[348]:
<class 'pandas.core.frame.DataFrame'>
Index: 1216 entries, 'burbs, The (1989) to eXistenZ (1999)
Data columns:
F     1216  non-null values
M     1216  non-null values
dtypes: float64(2)
```

Чтобы найти фильмы, оказавшиеся на первом месте у зрителей-женщин, мы можем отсортировать результат по столбцу `F` в порядке убывания:

```
In [350]: top_female_ratings = mean_ratings.sort_index(by='F', ascending=False)

In [351]: top_female_ratings[:10]
Out[351]:
gender                                                F         M
Close Shave, A (1995)                          4.644444  4.473795
Wrong Trousers, The (1993)                     4.588235  4.478261
```

```
Sunset Blvd. (a.k.a. Sunset Boulevard) (1950)            4.572650  4.464589
Wallace & Gromit: The Best of Aardman Animation (1996)   4.563107  4.385075
Schindler's List (1993)                                  4.562602  4.491415
Shawshank Redemption, The (1994)                         4.539075  4.560625
Grand Day Out, A (1992)                                  4.537879  4.293255
To Kill a Mockingbird (1962)                             4.536667  4.372611
Creature Comforts (1990)                                 4.513889  4.272277
Usual Suspects, The (1995)                               4.513317  4.518248
```

Измерение несогласия в оценках

Допустим, мы хотим найти фильмы, по которым мужчины и женщины сильнее всего разошлись в оценках. Для этого можно добавить столбец `mean_ratings`, содержащий разность средних, а затем отсортировать по нему:

```
In [352]: mean_ratings['diff'] = mean_ratings['M'] - mean_ratings['F']
```

Сортировка по столбцу `'diff'` дает фильмы с наибольшей разностью оценок, которые больше нравятся женщинам:

```
In [353]: sorted_by_diff = mean_ratings.sort_index(by='diff')

In [354]: sorted_by_diff[:15]
Out[354]:
gender                                     F         M       diff
Dirty Dancing (1987)                3.790378  2.959596  -0.830782
Jumpin' Jack Flash (1986)           3.254717  2.578358  -0.676359
Grease (1978)                       3.975265  3.367041  -0.608224
Little Women (1994)                 3.870588  3.321739  -0.548849
Steel Magnolias (1989)              3.901734  3.365957  -0.535777
Anastasia (1997)                    3.800000  3.281609  -0.518391
Rocky Horror Picture Show, The (1975)  3.673016  3.160131  -0.512885
Color Purple, The (1985)            4.158192  3.659341  -0.498851
Age of Innocence, The (1993)        3.827068  3.339506  -0.487561
Free Willy (1993)                   2.921348  2.438776  -0.482573
French Kiss (1995)                  3.535714  3.056962  -0.478752
Little Shop of Horrors, The (1960)  3.650000  3.179688  -0.470312
Guys and Dolls (1955)               4.051724  3.583333  -0.468391
Mary Poppins (1964)                 4.197740  3.730594  -0.467147
Patch Adams (1998)                  3.473282  3.008746  -0.464536
```

Изменив порядок строк на противоположный и снова отобрав первые 15 строк, мы получим фильмы, которым мужчины поставили высокие, а женщины – низкие оценки:

```
# Изменяем порядок строк на противоположный и отбираем первые 15 строк
In [355]: sorted_by_diff[::-1][:15]
Out[355]:

gender                                     F         M      diff
Good, The Bad and The Ugly, The (1966) 3.494949  4.221300  0.726351
Kentucky Fried Movie, The (1977)       2.878788  3.555147  0.676359
Dumb & Dumber (1994)                   2.697987  3.336595  0.638608
```

```
Longest Day, The (1962)                3.411765  4.031447  0.619682
Cable Guy, The (1996)                  2.250000  2.863787  0.613787
Evil Dead II (Dead By Dawn) (1987)     3.297297  3.909283  0.611985
Hidden, The (1987)                     3.137931  3.745098  0.607167
Rocky III (1982)                       2.361702  2.943503  0.581801
Caddyshack (1980)                      3.396135  3.969737  0.573602
For a Few Dollars More (1965)          3.409091  3.953795  0.544704
Porky's (1981)                         2.296875  2.836364  0.539489
Animal House (1978)                    3.628906  4.167192  0.538286
Exorcist, The (1973)                   3.537634  4.067239  0.529605
Fright Night (1985)                    2.973684  3.500000  0.526316
Barb Wire (1996)                       1.585366  2.100386  0.515020
```

А теперь допустим, что нас интересуют фильмы, вызвавшие наибольшее разногласие у зрителей независимо от пола. Разногласие можно изменить с помощью дисперсии или стандартного отклонения оценок:

```
# Стандартное отклонение оценок, сгруппированных по названию
In [356]: rating_std_by_title = data.groupby('title')['rating'].std()

# Оставляем только active_titles
In [357]: rating_std_by_title = rating_std_by_title.ix[active_titles]

# Упорядочиваем Series по значению в порядке убывания
In [358]: rating_std_by_title.order(ascending=False)[:10]
Out[358]:
title
Dumb & Dumber (1994)                   1.321333
Blair Witch Project, The (1999)        1.316368
Natural Born Killers (1994)            1.307198
Tank Girl (1995)                       1.277695
Rocky Horror Picture Show, The (1975)  1.260177
Eyes Wide Shut (1999)                  1.259624
Evita (1996) 1.253631
Billy Madison (1995)                   1.249970
Fear and Loathing in Las Vegas (1998)  1.246408
Bicentennial Man (1999)                1.245533
Name: rating
```

Вы, наверное, обратили внимание, что жанры фильма разделяются вертикальной чертой (|). Чтобы провести анализ по жанрам, прошлось бы проделать дополнительную работу по преобразованию данных в более удобную форму. Ниже я еще вернусь к этому набору данных и покажу, как это сделать.

Имена, которые давали детям в США за период с 1880 по 2010 год

Управление социального обеспечения США выложило в сеть данные о частоте встречаемости детских имен за период с 1880 года по настоящее время. Хэдли Уикхэм (Hadley Wickham), автор нескольких популярных пакетов для R, часто использует этот пример для иллюстрации манипуляций с данными в R.

```
In [4]: names.head(10)
Out[4]:
          name  sex  births  year
0        Mary    F    7065  1880
1        Anna    F    2604  1880
2        Emma    F    2003  1880
3   Elizabeth    F    1939  1880
4      Minnie    F    1746  1880
5    Margaret    F    1578  1880
6         Ida    F    1472  1880
7       Alice    F    1414  1880
8      Bertha    F    1320  1880
9       Sarah    F    1288  1880
```

С этим набором можно проделать много интересного.

- Наглядно представить долю младенцев, получавших данное имя (совпадающее с вашим или какое-нибудь другое) за весь период времени.
- Определить относительный ранг имени.
- Найти самые популярные в каждом году имена или имена, для которых фиксировалось наибольшее увеличение или уменьшение частоты.
- Проанализировать тенденции выбора имен: количество гласных и согласных, длину, общее разнообразие, изменение в написании, первые и последние буквы.
- Проанализировать внешние источники тенденций: библейские имена, имена знаменитостей, демографические изменения.

С помощью уже рассмотренных инструментов большая часть этих задач решается очень просто, и я это кратко продемонстрирую. Призываю вас скачать и исследовать этот набор данных самостоятельно. Если вы обнаружите интересную закономерность, буду рад узнать про нее.

На момент написания этой книги Управление социального обеспечения США представило данные в виде набора файлов, по одному на каждый год, в которых указано общее число родившихся младенцев для каждой пары пол/имя. Архив этих файлов находится по адресу

```
http://www.ssa.gov/oact/babynames/limits.html
```

Если со временем адрес этой страницы поменяется, найти ее, скорее всего, можно будет с помощью поисковой системы. Загрузив и распаковав файл `names.zip`, вы получите каталог, содержащий файлы с именами вида `yob1880.txt`. С помощью команды UNIX `head` я могу вывести первые 10 строк каждого файла (в Windows можно воспользоваться командой `more` или открыть файл в текстовом редакторе):

```
In [367]: !head -n 10 names/yob1880.txt
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
```

```
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

Поскольку поля разделены запятыми, файл можно загрузить в объект DataFrame методом `pandas.read_csv`:

```
In [368]: import pandas as pd

In [369]: names1880 = pd.read_csv('names/yob1880.txt', names=['name', 'sex', 'births'])

In [370]: names1880
Out[370]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2000 entries, 0 to 1999
Data columns:
name       2000  non-null values
sex        2000  non-null values
births     2000  non-null values
dtypes: int64(1), object(2)
```

В эти файлы включены только имена, которыми были названы не менее 5 младенцев в году, поэтому для простоты сумму значений в столбце sex можно считать общим числом родившихся в данном году младенцев:

```
In [371]: names1880.groupby('sex').births.sum()
Out[371]:
sex
F        90993
M       110493
Name: births
```

Поскольку в каждом файле находятся данные только за один год, то первое, что нужно сделать, – собрать все данные в единый объект DataFrame и добавить поле `year`. Это легко сделать методом `pandas.concat`:

```
# На данный момент 2010 - последний доступный год
years = range(1880, 2011)

pieces = []
columns = ['name', 'sex', 'births']

for year in years:
 path = 'names/yob%d.txt' % year
 frame = pd.read_csv(path, names=columns)

 frame['year'] = year
 pieces.append(frame)

 # Собрать все данные в один объект DataFrame
 names = pd.concat(pieces, ignore_index=True)
```

Обратим внимание на два момента. Во-первых, напомним, что `concat` по умолчанию объединяет объекты DataFrame построчно. Во-вторых, следует задать параметр `ignore_index=True`, потому что нам неинтересно сохранять исходные номера строк, прочитанных методом `read_csv`. Таким образом, мы получили очень большой DataFrame, содержащий данные обо всех именах.

Выглядит объект `names` следующим образом:

```
In [373]: names
Out[373]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name      1690784  non-null values
sex       1690784  non-null values
births    1690784  non-null values
year      1690784  non-null values
dtypes: int64(2), object(2)
```

Имея эти данные, мы уже можем приступить к агрегированию на уровне года и пола, используя метод `groupby` или `pivot_table` (см. рис. 2.4):

```
In [374]: total_births = names.pivot_table('births', rows='year',
   .....:                                  cols='sex', aggfunc=sum)

In [375]: total_births.tail()
Out[375]:
sex         F        M
year
2006  1896468  2050234
2007  1916888  2069242
2008  1883645  2032310
2009  1827643  1973359
2010  1759010  1898382
In [376]: total_births.plot(title='Total births by sex and year')
```

Далее вставим столбец `prop`, содержащий долю младенцев, получивших данное имя, относительно общего числа родившихся. Значение `prop`, равное `0.02`, означает, что данное имя получили 2 из 100 младенцев. Затем сгруппируем данные по году и полу и добавим в каждую группу новый столбец:

```
def add_prop(group):
 # При целочисленном делении производится округление с недостатком
 births = group.births.astype(float)

 group['prop'] = births / births.sum()
 return group
names = names.groupby(['year', 'sex']).apply(add_prop)
```

Напомним, что поскольку тип поля `births` – целое, для вычисления дробного числа необходимо привести числитель или знаменатель к типу с плавающей точкой (если только вы не работаете с Python 3!).

Рис. 2.4. Общее количество родившихся по полу и году

Получившийся в результате пополненный набор данных состоит из таких столбцов:

```
In [378]: names
Out[378]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1690784 entries, 0 to 1690783
Data columns:
name    1690784  non-null values
sex     1690784  non-null values
births  1690784  non-null values
year    1690784  non-null values
prop    1690784  non-null values
dtypes: float64(1), int64(2), object(2)
```

При выполнении такой операции группировки часто бывает полезно произвести проверку разумности результата, например, удостовериться, что сумма значений в столбце `prop` по всем группам равна 1. Поскольку это данные с плавающей точкой, воспользуемся методом `np.allclose`, который проверяет, что сумма по группам достаточно близка к 1 (хотя может и не быть равна в точности).

```
In [379]: np.allclose(names.groupby(['year', 'sex']).prop.sum(), 1)
Out[379]: True
```

Далее я извлеку подмножество данных, чтобы упростить последующий анализ: первые 1000 имен для каждой комбинации пола и года. Это еще одна групповая операция:

```
def get_top1000(group):
 return group.sort_index(by='births', ascending=False)[:1000]
grouped = names.groupby(['year', 'sex'])
top1000 = grouped.apply(get_top1000)
```

Если вы предпочитаете все делать самостоятельно, то можно поступить и так:

```
pieces = []
for year, group in names.groupby(['year', 'sex']):
  pieces.append(group.sort_index(by='births', ascending=False)[:1000])
top1000 = pd.concat(pieces, ignore_index=True)
```

Теперь результирующий набор стал заметно меньше:

```
In [382]: top1000
Out[382]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 261877 entries, 0 to 261876
Data columns:
name    261877  non-null values
sex     261877  non-null values
births  261877  non-null values
year    261877  non-null values
prop    261877  non-null values
dtypes: float64(1), int64(2), object(2)
```

Этот набор, содержащий первые 1000 записей, мы и будем использовать для исследования данных в дальнейшем.

Анализ тенденций в выборе имен

Имея полный набор данных и первые 1000 записей, мы можем приступить к анализу различных интересных тенденций. Для начала решим простую задачу: разобьем набор Top 1000 на части, относящиеся к мальчикам и девочкам.

```
In [383]: boys = top1000[top1000.sex == 'M']

In [384]: girls = top1000[top1000.sex == 'F']
```

Можно нанести на график простые временные ряды, например количество Джонов и Мэри в каждом году, но для этого потребуется предварительное переформатирование. Сформируем сводную таблицу, в которой представлено общее число родившихся по годам и по именам:

```
In [385]: total_births = top1000.pivot_table('births', rows='year', cols='name',
    .....:                                    aggfunc=sum)
```

Теперь можно нанести на график несколько имен, воспользовавшись методом `plot` объекта DataFrame:

```
In [386]: total_births
Out[386]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 131 entries, 1880 to 2010
Columns: 6865 entries, Aaden to Zuri
dtypes: float64(6865)

In [387]: subset = total_births[['John', 'Harry', 'Mary', 'Marilyn']]
```

```
In [388]: subset.plot(subplots=True, figsize=(12, 10), grid=False,
   .....:             title="Number of births per year")
```

Результат показан на рис. 2.5. Глядя на него, можно сделать вывод, что эти имена в Америке вышли из моды. Но на самом деле картина несколько сложнее, как станет ясно в следующем разделе.

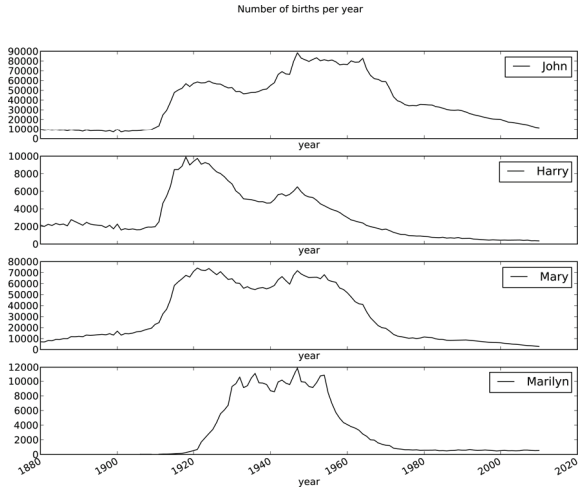


Рис. 2.5. Распределение нескольких имен мальчиков и девочек по годам

Измерение роста разнообразия имен

Убывание кривых на рисунках выше можно объяснить тем, что меньше родителей стали выбирать такие распространенные имена. Эту гипотезу можно проверить и подтвердить имеющимися данными. Один из возможных показателей – доля родившихся в наборе 1000 самых популярных имен, который я агрегирую по году и полу:

```
In [390]: table = top1000.pivot_table('prop', rows='year',
   .....:                             cols='sex', aggfunc=sum)

In [391]: table.plot(title='Sum of table1000.prop by year and sex',
   .....:       yticks=np.linspace(0, 1.2, 13), xticks=range(1880, 2020, 10))
```

Результат показан на рис. 2.6. Действительно, похоже, что разнообразие имен растет (доля в первой тысяче падает). Другой интересный показатель – количество различных имен среди первых 50 % родившихся, упорядоченное по популярности в порядке убывания. Вычислить его несколько сложнее. Рассмотрим только имена мальчиков, родившихся в 2010 году:

```
In [392]: df = boys[boys.year == 2010]

In [393]: df
```

```
Out[393]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 260877 to 261876
Data columns:
name    1000  non-null values
sex     1000  non-null values
births  1000  non-null values
year    1000  non-null values
prop    1000  non-null values
dtypes: float64(1), int64(2), object(2)
```

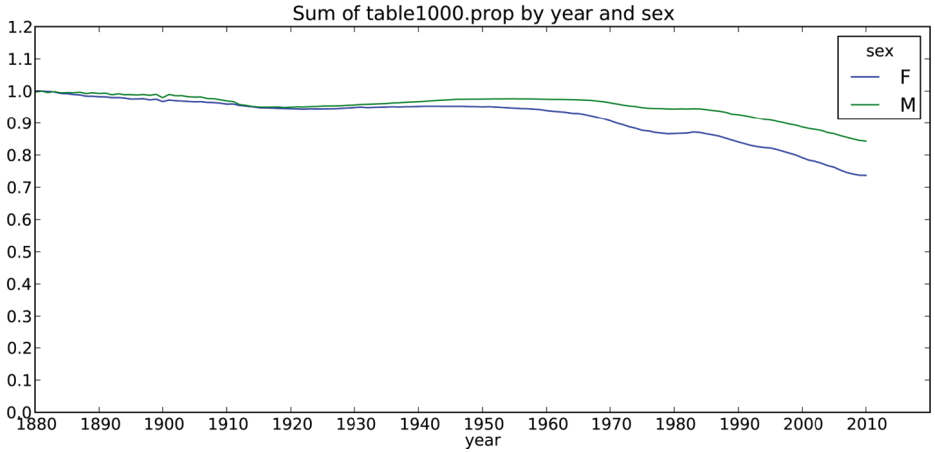


Рис. 2.6. Доля родившихся мальчиков и девочек, представленных в первой тысяче имен

После сортировки `prop` в порядке убывания мы хотим узнать, сколько популярных имен нужно взять, чтобы достичь 50 %. Можно написать для этого цикл `for`, но NumPy предлагает более хитроумный векторный подход. Если вычислить накопительные суммы `cumsum` массива `prop`, а затем вызвать метод `searchsorted`, то будет возвращена позиция в массиве накопительных сумм, в которую нужно было бы вставить `0.5`, чтобы не нарушить порядок сортировки:

```
In [394]: prop_cumsum = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [395]: prop_cumsum[:10]
Out[395]:
260877   0.011523
260878   0.020934
260879   0.029959
260880   0.038930
260881   0.047817
260882   0.056579
260883   0.065155
260884   0.073414
260885   0.081528
```

```
260886   0.089621

In [396]: prop_cumsum.searchsorted(0.5)
Out[396]: 116
```

Поскольку индексация массивов начинается с нуля, то нужно прибавить к результату 1 – получится 117. Заметим, что в 1900 году этот показатель был гораздо меньше:

```
In [397]: df = boys[boys.year == 1900]

In [398]: in1900 = df.sort_index(by='prop', ascending=False).prop.cumsum()

In [399]: in1900.searchsorted(0.5) + 1
Out[399]: 25
```

Теперь нетрудно применить эту операцию к каждой комбинации года и пола; произведем группировку по этим полям с помощью метода `groupby`, а затем с помощью метода `apply` применим функцию, возвращающую нужный показатель для каждой группы:

```
def get_quantile_count(group, q=0.5):
  group = group.sort_index(by='prop', ascending=False)
  return group.prop.cumsum().searchsorted(q) + 1

diversity = top1000.groupby(['year', 'sex']).apply(get_quantile_count)
diversity = diversity.unstack('sex')
```

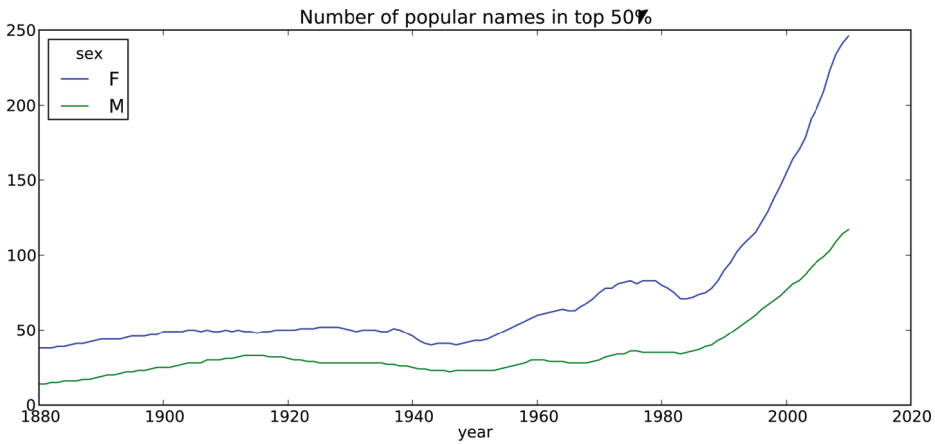


Рис. 2.7. График зависимости разнообразия от года

В получившемся объекте DataFrame с именем `diversity` хранится два временных ряда, по одному для каждого поля, индексированные по году. Его можно исследовать в IPython и, как и раньше, нанести на график (рис. 2.7).

```
In [401]: diversity.head()
Out[401]:
sex    F   M
year
1880  38  14
1881  38  14
1882  38  15
1883  39  15
1884  39  16

In [402]: diversity.plot(title="Number of popular names in top 50%")
```

Как видим, девочкам всегда давали более разнообразные имена, чем мальчикам, и со временем эта тенденция проявляется все ярче. Анализ того, что именно является причиной разнообразия, например рост числа вариантов написания одного и того же имени, оставляю читателю.

Революция «последней буквы»

В 2007 году исследовательница детских имен Лаура Уоттенберг (Laura Wattenberg) отметила на своем сайте (*http://www.babynamewizard.com*), что распределение имен мальчиков по последней букве за последние 100 лет существенно изменилось. Чтобы убедиться в этом, я сначала агрегирую данные полного набора обо всех родившихся по году, полу и последней букве:

```
# извлекаем последнюю букву имени в столбце name
get_last_letter = lambda x: x[-1]
last_letters = names.name.map(get_last_letter)
last_letters.name = 'last_letter'

table = names.pivot_table('births', rows=last_letters,
                          cols=['sex', 'year'], aggfunc=sum)
```

Затем выберу из всего периода три репрезентативных года и напечатаю первые несколько строк:

```
In [404]: subtable = table.reindex(columns=[1910, 1960, 2010], level='year')

In [405]: subtable.head()
Out[405]:
sex               F                            M
year           1910    1960    2010   1910    1960    2010
last_letter
a            108376  691247  670605    977    5204   28438
b               NaN     694     450    411    3912   38859
c                 5      49     946    482   15476   23125
d              6750    3729    2607  22111  262112   44398
e            133569  435013  313833  28655  178823  129012
```

Далее я пронормирую эту таблицу на общее число родившихся, чтобы вычислить новую таблицу, содержащую долю от общего родившихся для каждого пола и каждой последней буквы:

```
In [406]: subtable.sum()
Out[406]:
sex  year
F    1910    396416
     1960   2022062
     2010   1759010
M    1910    194198
     1960   2132588
     2010   1898382

In [407]: letter_prop = subtable / subtable.sum().astype(float)
```

Зная доли букв, я теперь могу нарисовать столбчатые диаграммы для каждого пола, разбив их по годам. Результат показан на рис. 2.8.

```
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 1, figsize=(10, 8))
letter_prop['M'].plot(kind='bar', rot=0, ax=axes[0], title='Male')
letter_prop['F'].plot(kind='bar', rot=0, ax=axes[1], title='Female',
                      legend=False)
```

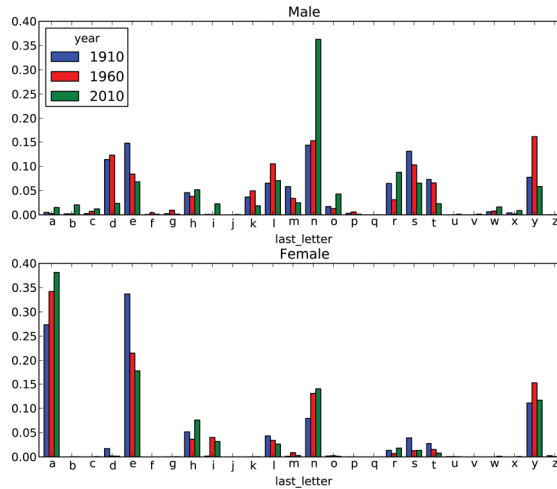


Рис. 2.8. Доли имен мальчиков и девочек, заканчивающихся на каждую букву

Как видим, с 1960-х годов доля имен мальчиков, заканчивающихся буквой «*n*», значительно возросла. Снова вернусь к созданной ранее полной таблице, пронормирую ее по году и полу, выберу некое подмножество букв для имен мальчиков и транспонирую, чтобы превратить каждый столбец во временной ряд:

```
In [410]: letter_prop = table / table.sum().astype(float)

In [411]: dny_ts = letter_prop.ix[['d', 'n', 'y'], 'M'].T

In [412]: dny_ts.head()
```

```
Out[412]:

                d         n         y
year
1880  0.083055  0.153213  0.075760
1881  0.083247  0.153214  0.077451
1882  0.085340  0.149560  0.077537
1883  0.084066  0.151646  0.079144
1884  0.086120  0.149915  0.080405
```

Имея этот объект DataFrame, содержащие временные ряды, я могу все тем же методом `plot` построить график изменения тенденций в зависимости от времени (рис. 2.9):

```
In [414]: dny_ts.plot()
```

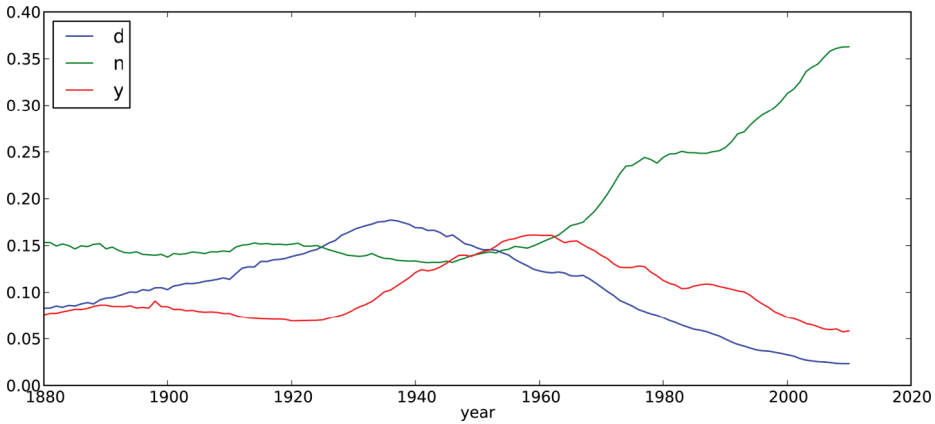


Рис. 2.9. Зависимость доли мальчиков с именами, заканчивающимися на буквы d, n, y, от времени

Мужские имена, ставшие женскими, и наоборот

Еще одно интересное упражнение – изучить имена, которые раньше часто давали мальчикам, а затем «сменили пол». Возьмем, к примеру, имя Lesley или Leslie. По набору `top1000` вычисляю список имен, начинающихся с `'lesl'`:

```
In [415]: all_names = top1000.name.unique()

In [416]: mask = np.array(['lesl' in x.lower() for x in all_names])

In [417]: lesley_like = all_names[mask]

In [418]: lesley_like
Out[418]: array([Leslie, Lesley, Leslee, Lesli, Lesly], dtype=object)
```

Далее можно оставить только эти имена и просуммировать количество родившихся, сгруппировав по имени, чтобы найти относительные частоты:

```
In [419]: filtered = top1000[top1000.name.isin(lesley_like)]

In [420]: filtered.groupby('name').births.sum()
Out[420]:
name
Leslee     1082
Lesley    35022
Lesli       929
Leslie   370429
Lesly     10067
Name: births
```

Затем агрегируем по полу и году и нормируем в пределах каждого года:

```
In [421]: table = filtered.pivot_table('births', rows='year',
    .....:                             cols='sex', aggfunc='sum')

In [422]: table = table.div(table.sum(1), axis=0)

In [423]: table.tail()
Out[423]:
sex   F   M
year
2006  1 NaN
2007  1 NaN
2008  1 NaN
2009  1 NaN
2010  1 NaN
```

Наконец, нетрудно построить график распределения по полу в зависимости от времени (рис. 2.10).

```
In [425]: table.plot(style={'M': 'k-', 'F': 'k--'})
```

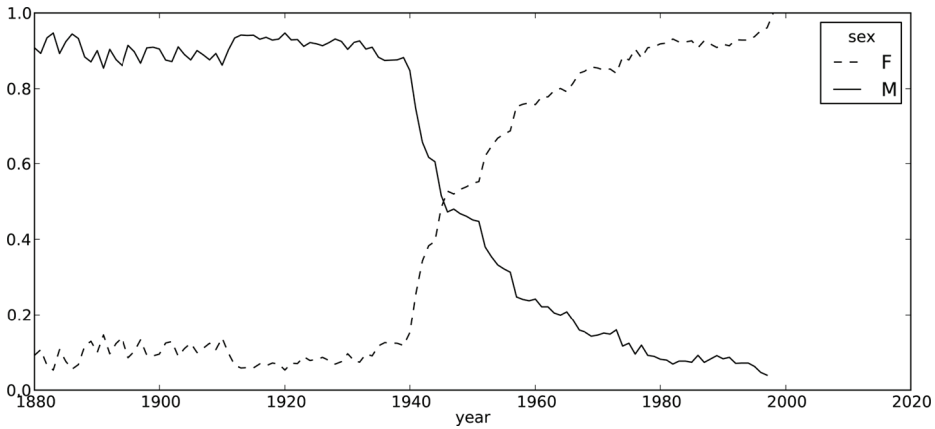


Рис. 2.10. Изменение во времени доли мальчиков и девочек с именами, похожими на Lesley

Выводы и перспективы

Приведенные в этой главе примеры довольно просты, но они позволяют составить представление о том, чего ожидать от последующих глав. Эта книга посвящена, прежде всего, инструментам, а не демонстрации более сложных аналитических методов. Овладев описываемыми приемами, вы сможете без труда реализовать собственные методы анализа (если, конечно, знаете, чего хотите!).

ГЛАВА 3. IPython: интерактивные вычисления и среда разработки

Действуй, не делая. Работай, не прилагая усилий. Думай о малом, как о большом, и о нескольких, как о многих. Работай над трудной задачей, пока она еще проста; реализуй великую цель через множество небольших шагов.

Лао Цзы

Меня часто спрашивают: «В какой среде разработки на Python вы работаете?». Почти всегда я отвечаю «IPython и текстовый редактор». Вы можете заменить текстовый редактор интегрированной средой разработки (IDE), если хотите иметь графические инструменты и средства автоматического завершения кода. Но и в этом случае я советую не отказываться от IPython. Некоторые IDE даже включают интеграцию с IPython, так что вы получаете лучшее из обоих миров.

Проект *IPython* в 2001 году основал Фернандо Перес как побочный продукт по ходу создания усовершенствованного интерактивного интерпретатора Python. Впоследствии он превратился в один из самых важных инструментов в арсенале ученых, работающих на Python. Сам по себе он не предлагает ни вычислительных, ни аналитических средств, но изначально спроектирован с целью повысить продуктивность интерактивных вычислений и разработки ПО. В его основе лежит последовательность действий «выполни и посмотри» вместо типичной для многих языков «отредактируй, откомпилируй и запусти». Он также очень тесно интегрирован с оболочкой операционной системы и с файловой системой. Поскольку анализ данных подразумевает исследовательскую работу, применение метода проб и ошибок и итеративный подход, то IPython почти во всех случаях позволяет ускорить выполнение работы.

Разумеется, сегодняшний IPython – это куда больше, чем просто усовершенствованная интерактивная оболочка Python. В его состав входит развитая графическая консоль с встроенными средствами построения графиков, интерактивный веб-блокнот и облегченный движок для быстрых параллельных вычислений. И подобно многим другим инструментам, созданным программистами для программистов, он настраивается в очень широких пределах. Некоторые из вышеупомянутых возможностей будут рассмотрены ниже.

Поскольку интерактивность неотъемлема от IPython, некоторые описываемые далее возможности трудно в полной мере продемонстрировать без «живой» консоли. Если вы читаете об IPython впервые, я рекомендую параллельно прорабатывать примеры, чтобы понять, как все это работает на практике. Как и в любой среде, основанной на активном использовании клавиатуры, для полного освоения инструмента необходимо запомнить комбинации клавиш для наиболее употребительных команд.

При первом чтении многие части этой главы (к примеру, профилирование и отладка) можно пропустить без ущерба для понимания последующего материала. Эта глава задумана как независимый, достаточно полный обзор функциональности IPython.

Основы IPython

IPython можно запустить из командной строки, как и стандартный интерпретатор Python, только для этого служит команда `ipython`:

```
$ ipython
Python 2.7.2 (default, May 27 2012, 21:26:12)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: a = 5

In [2]: a
Out[2]: 5
```

Чтобы выполнить произвольное предложение Python, нужно ввести его и нажать клавишу **Enter**. Если ввести только имя переменной, то IPython выведет строковое представление объекта:

```
In [542]: data = {i : randn() for i in range(7)}

In [543]: data
Out[543]:
{0: 0.6900018528091594,
 1: 1.0015434424937888,
 2: -0.5030873913603446,
 3: -0.6222742250596455,
 4: -0.9211686080130108,
 5: -0.726213492660829,
 6: 0.2228955458351768}
```

Многие объекты Python форматируются для удобства чтения; такая *красивая печать* отличается от обычного представления методом `print`. Тот же словарь, напечатанный в стандартном интерпретаторе Python, выглядел бы куда менее презентабельно:

```
>>> from numpy.random import randn
>>> data = {i : randn() for i in range(7)}
>>> print data
{0: -1.5948255432744511, 1: 0.10569006472787983, 2: 1.972367135977295,
3: 0.15455217573074576, 4: -0.24058577449429575, 5: -1.2904897053651216,
6: 0.3308507317325902}
```

IPython предоставляет также средства для исполнения произвольных блоков кода (путем копирования и вставки) и целых Python-скриптов. Эти вопросы будут рассмотрены чуть ниже.

Завершение по нажатию клавиши Tab

На первый взгляд, оболочка IPython очень похожа на стандартный интерпретатор Python с мелкими косметическими изменениями. Пользователям программы Mathematica знакомы пронумерованные строки ввода и вывода. Одно из существенных преимуществ над стандартной оболочкой Python – завершение по нажатию клавиши **Tab**, функция, реализованная в большинстве интерактивных сред анализа данных. Если во время ввода выражения нажать `<Tab>`, то оболочка произведет поиск в пространстве имен всех переменных (объектов, функций и т. д.), имена которых начинаются с введенной к этому моменту строки:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple and an_example any
```

Обратите внимание, что IPython вывел обе определенные выше переменные, а также ключевое слово Python `and` и встроенную функцию `any`. Естественно, можно также завершать имена методов и атрибутов любого объекта, если предварительно ввести точку:

```
In [3]: b = [1, 2, 3]

In [4]: b.<Tab>
b.append b.extend b.insert b.remove b.sort
b.count b.index b.pop b.reverse
```

То же самое относится и к модулям:

```
In [1]: import datetime

In [2]: datetime.<Tab>
datetime.date           datetime.MAXYEAR      datetime.timedelta
```

```
datetime.datetime       datetime.MINYEAR      datetime.tzinfo
datetime.datetime_CAPI  datetime.time
```

Отметим, что IPython по умолчанию скрывает методы и атрибуты, начинающиеся знаком подчеркивания, например магические методы и внутренние «закрытые» методы и атрибуты, чтобы не загромождать экран (и не смущать неопытных пользователей). На них автозавершение также распространяется, нужно только сначала набрать знак подчеркивания. Если вы предпочитаете всегда видеть такие методы при автозавершении, измените соответствующий режим в конфигурационном файле IPython.

Завершение по нажатию **Tab** работает во многих контекстах, помимо поиска в интерактивном пространстве имен и завершения атрибутов объекта или модуля. Если нажать `<Tab>` при вводе чего-то, похожего на путь к файлу (даже внутри строки Python), то будет произведен поиск в файловой системе:

```
In [3]: book_scripts/<Tab>
book_scripts/cprof_example.py        book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py          book_scripts/prof_mod.py

In [3]: path = 'book_scripts/<Tab>
book_scripts/cprof_example.py        book_scripts/ipython_script_test.py
book_scripts/ipython_bug.py          book_scripts/prof_mod.py
```

В сочетании с командой `%run` (см. ниже) эта функция несомненно позволит вам меньше лупить по клавиатуре.

Автозавершение позволяет также сэкономить время при вводе именованных аргументов функции (в том числе и самого знака `=`).

Интроспекция

Если ввести вопросительный знак (`?`) до или после имени переменной, то будет напечатана общая информация об объекте:

```
In [545]: b?
Type:        list
String Form: [1, 2, 3]
Length:      3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

Это называется *интроспекцией объекта*. Если объект представляет собой функцию или метод экземпляра, то будет показана строка документации, если она существует. Допустим, мы написали такую функцию:

```
def add_numbers(a, b):
  """
  Сложить два числа

  Возвращает
  ----------
```

```
  the_sum : тип аргументов
  """
  return a + b
```

Тогда при вводе знака ? мы увидим строку документации:

```
In [547]: add_numbers?
Type:        function
String Form: <function add_numbers at 0x5fad848>
File:        book_scripts/<ipython-input-546-5473012eeb65>
Definition:  add_numbers(a, b)
Docstring:
Сложить два числа
Возвращает
----------
the_sum : тип аргументов
```

Два вопросительных знака ?? покажут также исходный код функции, если это возможно:

```
In [548]: add_numbers??
Type:        function
String Form: <function add_numbers at 0x5fad848>
File:        book_scripts/<ipython-input-546-5473012eeb65>
Definition:  add_numbers(a, b)
Source:
def add_numbers(a, b):
  """
  Сложить два числа

  Возвращает
  ----------
  the_sum : тип аргументов
  """
  return a + b
```

И последнее применение ? – поиск в пространстве имен IPython по аналогии со стандартной командной строкой UNIX или Windows. Если ввести несколько символов в сочетании с метасимволом *, то будут показаны все имена по указанной маске. Например, вот как можно получить список всех функций в пространстве имен верхнего уровня NumPy, имена которых содержат строку load:

```
In [549]: np.*load*?
np.load
np.loads
np.loadtxt
np.pkgload
```

Команда %run

Команда %run позволяет выполнить любой файл как Python-программу в контексте текущего сеанса IPython. Предположим, что в файле ipython_script_test.py хранится такой простенький скрипт:

```
def f(x, y, z):
  return (x + y) / z

a = 5
b = 6
c = 7.5

result = f(a, b, c)
```

Этот скрипт можно выполнить, передав имя файла команде `%run`:

```
In [550]: %run ipython_script_test.py
```

Скрипт выполняется в *пустом пространстве имен* (в которое ничего не импортировано и в котором не определены никакие переменные), поэтому его поведение должно быть идентично тому, что получается при запуске программы из командной строки командой `python script.py`. Все переменные (импортированные, функции, глобальные объекты), определенные в файле (до момента исключения, если таковое произойдет), будут доступны оболочке IPython:

```
In [551]: c
Out[551]: 7.5

In [552]: result
Out[552]: 1.4666666666666666
```

Если Python-скрипт ожидает передачи аргументов из командной строки (которые должны попасть в массив `sys.argv`), то их можно перечислить после пути к файлу, как в командной строке.

Если вы хотите дать скрипту доступ к переменным, уже определенным в интерактивном пространстве имен IPython, используйте команду `%run -i`, а не просто `%run`.

Прерывание выполняемой программы

Нажатие `<Ctrl-C>` во время выполнения кода, запущенного с помощью `%run`, или просто долго работающей программы, приводит к возбуждению исключения `KeyboardInterrupt`. В этом случае почти все Python-программы немедленно прекращают работу, если только не возникло очень редкое стечение обстоятельств.

Если Python-код вызвал откомпилированный модуль расширения, то нажатие `<Ctrl-C>` не всегда приводит к немедленному завершению. В таких случаях нужно либо дождаться возврата управления интерпретатору Python, либо – если случилось что-то ужасное – принудительно снять процесс Python с помощью диспетчера задач ОС.

Исполнение кода из буфера обмена

Когда нужно быстро и без хлопот выполнить код в IPython, можно просто взять его из буфера обмена. На первый взгляд, неряшливо, но на практике очень полезно. Например, при разработке сложного или долго работающего приложения иногда желательно исполнять скрипт по частям, останавливаясь после каждого шага, чтобы проверить загруженные данные и результаты. Или, допустим, вы нашли какой-то фрагмент кода в Интернете и хотите поэкспериментировать с ним, не создавая новый `py`-файл.

Для извлечения фрагмента кода из буфера обмена во многих случаях достаточно нажать `<Ctrl-Shift-V>`. Отметим, что это не стопроцентно надежный способ, потому в таком режиме имитируется ввод каждой строки в IPython, а символы новой строки трактуются как нажатие `<Enter>`. Это означает, что если извлекается код, содержащий блок с отступом, и в нем присутствует пустая строка, то IPython будет считать, что блок закончился. При выполнении следующей строки в блоке возникнет исключение `IndentationError`. Например, такой код:

```
x = 5
y = 7
if x > 5:
  x += 1

  y = 8
```

после вставки из буфера обмена работать не будет:

```
In [1]: x = 5

In [2]: y = 7

In [3]: if x > 5:
   ...: x += 1
   ...:

In [4]: y = 8
IndentationError: unexpected indent

If you want to paste code into IPython, try the %paste and %cpaste
magic functions.
```

В сообщении об ошибке предлагается использовать магические функции `%paste` и `%cpaste`. Функция `%paste` принимает текст, находящийся в буфере обмена, и исполняет его в оболочке как единый блок:

```
In [6]: %paste
x = 5
y = 7
if x > 5:
  x += 1

  y = 8
## -- Конец вставленного текста --
```

В зависимости от платформы и способа установки Python может случиться, что функция `%paste` откажется работать, хотя это маловероятно. В пакетных дистрибутивах типа EPDFree (описан во введении) такой проблемы быть не должно.

Функция `%cpaste` аналогична, но выводит специальное приглашение для вставки кода:

```
In [7]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:x = 5
:y = 7
:if x > 5:
:  x += 1
:
:  y = 8
:--
```

При использовании `%cpaste` вы можете вставить сколько угодно кода, перед тем как начать его выполнение. Например, `%cpaste` может пригодиться, если вы хотите посмотреть на вставленный код до выполнения. Если окажется, что случайно вставлен не тот код, то из `%cpaste` можно выйти нажатием `<Ctrl-C>`.

Ниже мы познакомимся с HTML-блокнотом IPython, который выводит поблочный анализ на новый уровень – с помощью работающего в браузере блокнота с исполняемыми ячейками, содержащими код.

Взаимодействие IPython с редакторами и IDE

Для некоторых текстовых редакторов, например Emacs и vim, существуют расширения, позволяющие отправлять блоки кода напрямую из редактора в запущенную оболочку IPython. Для получения дополнительных сведений зайдите на сайт IPython или поищите в Интернете.

Для некоторых IDE существуют подключаемые модули, например PyDev для Eclipse или Python Tools для Visual Studio от Microsoft (а, возможно, и для других), обеспечивающие интеграцию с консольным приложением IPython. Если вы хотите работать в IDE, не отказываясь от консольных функций IPython, то это может стать удачным решением.

Комбинации клавиш

В IPython есть много комбинаций клавиш для навигации по командной строке (они знакомы пользователям текстового редактора Emacs или оболочки UNIX bash) и взаимодействия с историей команд (см. следующий раздел). В табл. 3.1 перечислены наиболее употребительные комбинации, а на рис. 3.1 некоторые из них, например перемещение курсора, проиллюстрированы.

Таблица 3.1. Стандартные комбинации клавиш IPython

Команда	Описание
`Ctrl-P` или `стрелка-вверх`	Просматривать историю команд назад в поисках команд, начинающихся с введенной строки
`Ctrl-N` или `стрелка-вниз`	Просматривать историю команд вперед в поисках команд, начинающихся с введенной строки
`Ctrl-R R`	Обратный поиск в истории в духе Readline (частичное соответствие)
`Ctrl-Shift-V`	Вставить текст из буфера обмена
`Ctrl-C`	Прервать исполнение программы
`Ctrl-A`	Переместить курсор в начало строки
`Ctrl-E`	Переместить курсор в конец строки
`Ctrl-K`	Удалить текст от курсора до конца строки
`Ctrl-U`	Отбросить весь текст в текущей строке
`Ctrl-F`	Переместить курсор на один символ вперед
`Ctrl-B`	Переместить курсор на один символ назад
`Ctrl-L`	Очистить экран

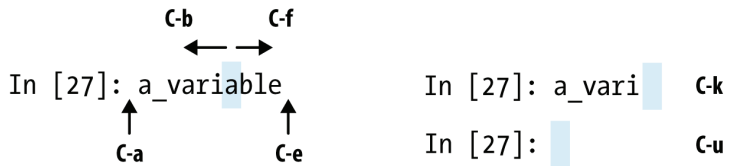


Рис. 3.1. Иллюстрация некоторых комбинаций клавиш IPython

Исключения и обратная трассировка

Если при выполнении любого предложения или скрипта, запущенного командой `%run`, возникнет исключение, то по умолчанию IPython напечатает все содержимое стека вызовов (обратную трассировку), сопроводив каждый вызов несколькими строками контекста.

```
In [553]: %run ch03/ipython_bug.py
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
/home/wesm/code/ipython/IPython/utils/py3compat.pyc in execfile(fname, *where)
  176 else:
  177 filename = fname
--> 178 __builtin__.execfile(filename, *where)
book_scripts/ch03/ipython_bug.py in <module>()
  13 throws_an_exception()
  14
```

```
---> 15 calling_things()
book_scripts/ch03/ipython_bug.py in calling_things()
  11 def calling_things():
  12 works_fine()
---> 13 throws_an_exception()
  14
  15 calling_things()
book_scripts/ch03/ipython_bug.py in throws_an_exception()
  7 a = 5
  8 b = 6
----> 9 assert(a + b == 10)
 10
 11 def calling_things():
AssertionError:
```

Наличие дополнительного контекста уже само по себе является большим преимуществом по сравнению со стандартным интерпретатором Python (который не выводит контекст). Объемом контекста можно управлять с помощью магической команды `%xmode` – от минимального (как в стандартном интерпретаторе Python) до подробного (когда включаются значения аргументов функции и другая информация). Ниже в этой главе мы увидим, что можно перемещаться по стеку (с помощью магических команд `%debug` и `%pdb`) после ошибки, это дает возможность производить посмертную отладку.

Магические команды

В IPython есть много специальных команд, называемых «магическими», цель которых – упростить решение типичных задач и облегчить контроль над поведением всей системы IPython. Магической называется команда, которой предшествует знак процента `%`. Например, магическая функция `%timeit` (мы подробно рассмотрим ее ниже) позволяет замерить время выполнения любого предложения Python, например умножения матриц:

```
In [554]: a = np.random.randn(100, 100)

In [555]: %timeit np.dot(a, a)
10000 loops, best of 3: 69.1 us per loop
```

Магические команды можно рассматривать как командные утилиты, исполняемые внутри IPython. У многих из них имеются дополнительные параметры «командной строки», список которых можно распечатать с помощью `?` (вы ведь так и думали, правда?)[1]:

```
In [1]: %reset?
Возвращает пространство имен в начальное состояние, удаляя все имена,
определенные пользователем.

Параметры
----------
```

[1] Сообщения выводятся на английском языке, но для удобства читателя переведены. – *Прим. перев.*

```
    -f : принудительная очистка без запроса подтверждения.

    -s : 'Мягкая' очистка: очищается только ваше пространство имен, а история
    остается. Ссылки на объекты можно удержать. По умолчанию (без этого
    параметра) выполняется 'жесткая' очистка, в результате чего вы получаете
    новый сеанс, а все ссылки на объекты в текущем сеансе удаляются.

Примеры
--------
In [6]: a = 1

In [7]: a
Out[7]: 1

In [8]: 'a' in _ip.user_ns
Out[8]: True

In [9]: %reset -f

In [1]: 'a' in _ip.user_ns
Out[1]: False
```

Магические функции по умолчанию можно использовать и без знака процента, если только нигде не определена переменная с таким же именем, как у магической функции. Этот режим называется *автомагическим*, его можно включить или выключить с помощью функции `%automagic`.

Поскольку к документации по IPython легко можно обратиться из системы, я рекомендую изучить все имеющиеся специальные команды, набрав `%quickref` или `%magic`. Я расскажу только о нескольких наиболее важных для продуктивной работы в области интерактивных вычислений и разработки в среде IPython.

Таблица 3.2. Часто используемые магические команды IPython

Команда	Описание
`%quickref`	Вывести краткую справку по IPython
`%magic`	Вывести подробную документацию по всем имеющимся магическим командам
`%debug`	Войти в интерактивный отладчик в точке последнего вызова, показанного в обратной трассировке исключения
`%hist`	Напечатать историю введенных команд (по желанию вместе с результатами)
`%pdb`	Автоматически входить в отладчик после любого исключения
`%paste`	Выполнить отформатированный Python-код, находящийся в буфере обмена
`%cpaste`	Открыть специальное приглашение для ручной вставки Python-кода, подлежащего выполнению

Команда	Описание
`%reset`	Удалить все переменные и прочие имена, определенные в интерактивном пространстве имен
`%page` *OBJECT*	Сформировать красиво отформатированное представление объекта и вывести его постранично
`%run` *script.py*	Выполнить Python-скрипт из IPython
`%prun` *предложение*	Выполнить *предложение* под управлением cProfile и вывести результаты профилирования
`%time` *предложение*	Показать время выполнения одного предложения
`%timeit` *предложение*	Выполнить предложение несколько раз и усреднить время выполнения. Полезно для хронометража кода, который выполняется очень быстро
`%who, %who_ls, %whos`	Вывести переменные, определенные в интерактивном пространстве имен, с различной степенью детализации
`%xdel` *переменная*	Удалить переменную и попытаться очистить все ссылки на объект во внутренних структурах данных IPython

Графическая консоль на базе Qt

Команда IPython разработала графическую консоль на базе библиотеки Qt (рис. 3.2), цель которой – скрестить возможность чисто консольного приложения со средствами, предоставляемыми виджетом обогащенного текста, в том числе встраиваемыми изображениями, режимом многострострочного редактирования и подсветкой синтаксиса. Если на вашей машине установлен пакет PyQt или PySide, то можно запустить приложение с встроенным построением графиков:

```
ipython qtconsole --pylab=inline
```

Qt-консоль позволяет запускать несколько процессов IPython в отдельных вкладках и тем самым переключаться с одной задачи на другую. Она также может разделять процесс с HTML-блокнотом IPython, о котором я расскажу ниже.

Интеграция с matplotlib и режим pylab

IPython так широко используется в научном сообществе отчасти потому, что он спроектирован как дополнение к библиотекам типа matplotlib и другим графическим инструментам. Если вы раньше никогда не работали с matplotlib, ничего страшного; ниже мы обсудим эту библиотеку во всех подробностях. Создав окно графика matplotlib в стандартной оболочке Python, вы будете неприятно поражены тем, что цикл обработки событий ГИП «перехватывает контроль» над сеансом Python до тех пор, пока окно не будет закрыто. Для интерактивного анализа данных и визуализации это не годится, поэтому в IPython реализована специальная логика для всех распространенных библиотек построения ГИП, так чтобы обеспечить органичную интеграцию с оболочкой.

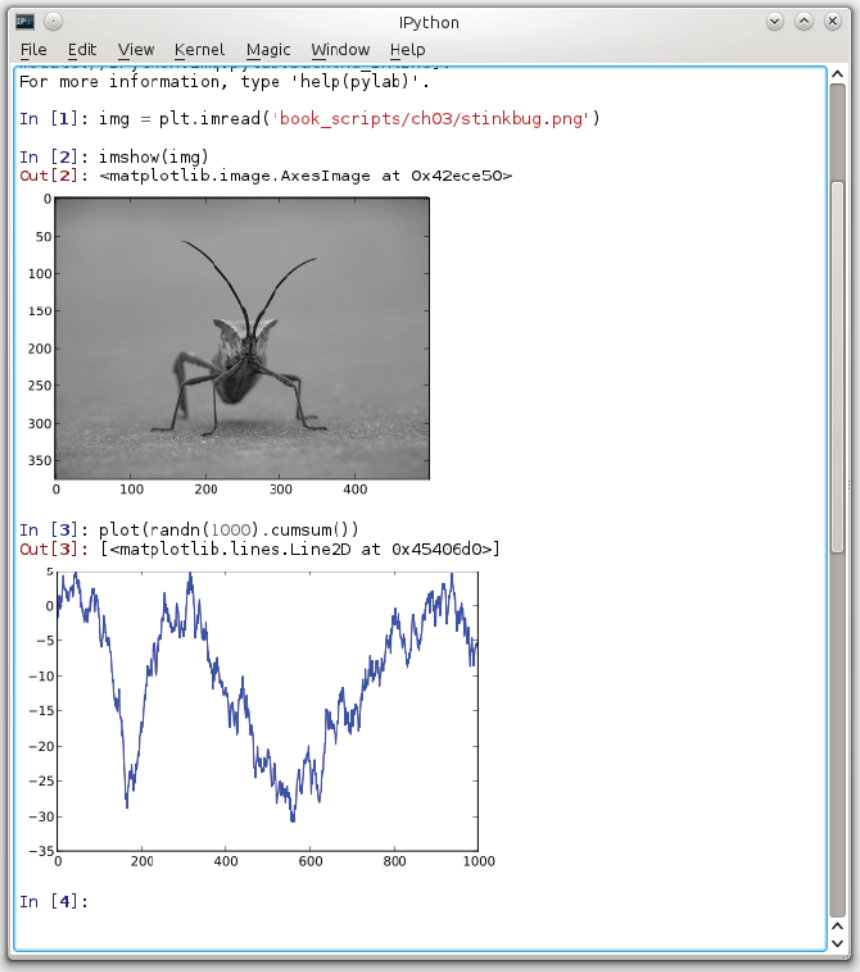


Рис. 3.2. Qt-консоль IPython

Для запуска IPython в режиме интеграции с matplotlib достаточно просто добавить флаг `--pylab` (дефисов должно быть два).

```
$ ipython --pylab
```

При этом произойдет несколько вещей. Во-первых, IPython запустится в режиме интеграции с ГИП по умолчанию, что позволит без проблем создавать окна графиков matplotlib. Во-вторых, в интерактивное пространство имен верхнего уровня будет импортирована большая часть NumPy и matplotlib, в результате чего создается среда интерактивных вычислений, напоминающая MATLAB и другие

предметно-ориентированные среды (рис. 3.3). Такую же конфигурацию можно задать и вручную, воспользовавшись командой `%gui` (наберите `%gui?`, чтобы узнать, как).

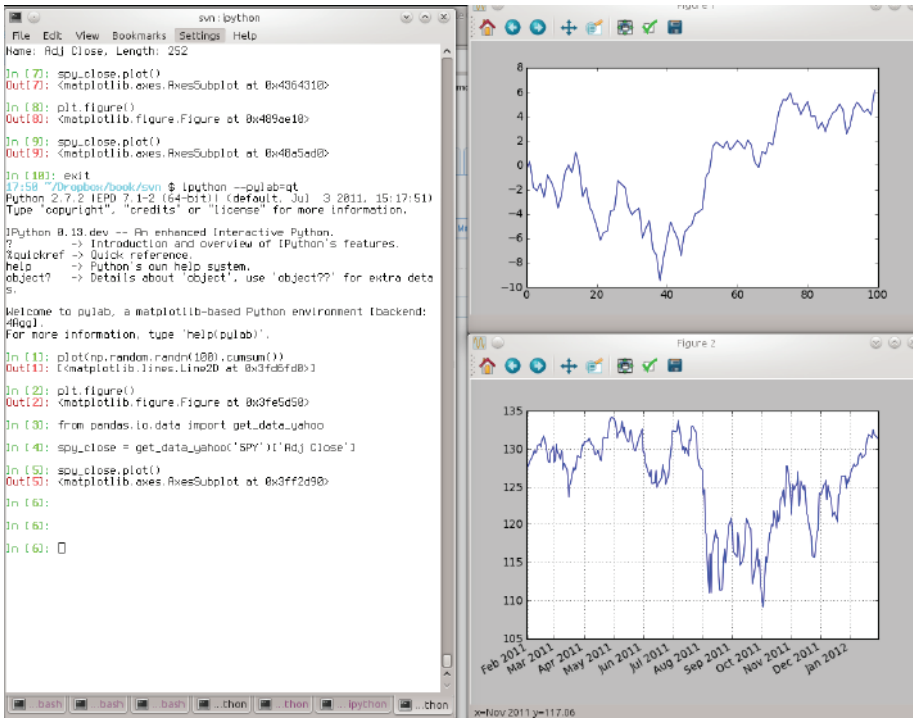


Рис. 3.3. Режим Pylab: IPython с окнами matplotlib

История команд

IPython поддерживает небольшую базу данных на диске, в которой хранятся тексты всех выполненных команд. Она служит нескольким целям:

- поиск, автозавершение и повторное выполнение ранее выполненных команд с минимальными усилиями;
- сохранение истории команд между сеансами;
- протоколирование истории ввода-вывода в файле.

Поиск в истории команд и повторное выполнение

Возможность искать и повторно выполнять предыдущие команды для многих является самой полезной функцией. Поскольку IPython рассчитан на итератив-

ную и интерактивную разработку кода, мы часто повторяем одни и те же команды, например `%run`. Допустим вы выполнили такую команду:

```
In[7]: %run first/second/third/data_script.py
```

и, ознакомившись с результатами работы скрипта (в предположении, что он завершился успешно), обнаружили ошибку в вычислениях. Разобравшись, в чем проблема, и исправив скрипт `data_script.py`, вы можете набрать несколько первых букв команды `%run` и нажать `<Ctrl-P>` или клавишу `<стрелка вверх>`. В ответ IPython найдет в истории команд первую из предшествующих команд, начинающуюся введенными буквами. При повторном нажатии `<Ctrl-P>` или `<стрелки вверх>` поиск будет продолжен. Если вы проскочили мимо нужной команды, ничего страшного. По истории команд можно перемещаться и *вперед* с помощью клавиш `<Ctrl-N>` или `<стрелка вниз>`. Стоит только попробовать, и вы начнете нажимать эти клавиши, не задумываясь.

Комбинация клавиш `<Ctrl-R>` дает ту же возможность частичного инкрементного поиска, что подсистема `readline`, применяемая в оболочках UNIX, например bash. В Windows функциональность `readline` реализуется самим IPython. Чтобы воспользоваться ей, нажмите `<Ctrl-R>`, а затем введите несколько символов, встречающихся в искомой строке ввода:

```
In [1]: a_command = foo(x, y, z)

(reverse-i-search)'com': a_command = foo(x, y, z)
```

Нажатие `<Ctrl-R>` приводит к циклическому просмотру истории в поисках строк, соответствующих введенным символам.

Входные и выходные переменные

Забыв присвоить результат вызова функции, вы можете горько пожалеть об этом. По счастью, IPython сохраняет ссылки как на входные команды (набранный вами текст), так и на выходные объекты в специальных переменных. Последний и предпоследний выходной объект хранятся соответственно в переменных _ (один подчерк) и __ (два подчерка):

```
In [556]: 2 ** 27
Out[556]: 134217728

In [557]: _
Out[557]: 134217728
```

Входные команды хранятся в переменных с именами вида `_iX`, где `X` – номер входной строки. Каждой такой входной переменной соответствует выходная переменная `_X`. Поэтому после ввода строки 27 будут созданы две новых переменных `_27` (для хранения выходного объекта) и `_i27` (для хранения входной команды).

```
In [26]: foo = 'bar'

In [27]: foo
```

```
Out[27]: 'bar'

In [28]: _i27
Out[28]: u'foo'

In [29]: _27
Out[29]: 'bar'
```

Поскольку входные переменные – это строки, то их можно повторно вычислить с помощью ключевого слова Python `exec`:

```
In [30]: exec _i27
```

Есть несколько магических функций, позволяющих работать с историей ввода и вывода. Функция `%hist` умеет показывать историю ввода полностью или частично, с номерами строк или без них. Функция `%reset` очищает интерактивное пространство имен и факультативно кэши ввода и вывода. Функция `%xdel` удаляет все ссылки на *конкретный* объект из внутренних структур данных IPython. Подробнее см. документацию по этим функциям.

Работая с очень большими наборами данных, имейте в виду, что объекты, хранящиеся в истории ввода-вывода IPython, не могут быть удалены из памяти сборщиком мусора – даже если вы удалите соответствующую переменную из интерактивного пространства имен встроенным оператором `del`. В таких случаях команды `%xdel` и `%reset` помогут избежать проблем с памятью.

Протоколирование ввода-вывода

IPython умеет протоколировать весь консольный сеанс, в том числе ввод и вывод. Режим протоколирования включается командой `%logstart`:

```
In [3]: %logstart
Activating auto-logging. Current session state plus future input saved.
Filename       : ipython_log.py
Mode           : rotate
Output logging : False
Raw input log  : False
Timestamping   : False
State          : active
```

Включить протоколирование IPython можно в любое время, и записан будет весь сеанс (в том числе и ранее выполненные команды). Таким образом, если в процессе работы вы решите сохранить все сделанное к этому моменту, достаточно будет включить протоколирование. Дополнительные параметры описаны в строке документации по функции `%logstart` (в частности, как изменить путь к файлу журнала) и связанным с ней функциям `%logoff`, `%logon`, `%logstate` и `%logstop`.

Взаимодействие с операционной системой

Еще одна важная особенность IPython — очень тесная интеграция с оболочкой операционной системы. Среди прочего это означает, что многие стандартные действия в командной строке можно выполнять в точности так же, как в оболочке Windows или UNIX (Linux, OS X), не выходя из IPython. Речь идет о выполнении команд оболочки, смене рабочего каталога и сохранении результатов команды в объекте Python (строке или списке). Существуют также простые средства для задания псевдонимов команд оболочки и создания закладок на каталоги.

Перечень магических функций и синтаксис вызова команд оболочки представлены в табл. 3.3. В следующих разделах я кратко расскажу о них.

Таблица 3.3. Команды IPython, относящиеся к операционной системе

Команда	Описание
`!cmd`	Выполнить команду в оболочке системы
`output = !cmd args`	Выполнить команду и сохранить в объекте `output` все выведенное на стандартный вывод
`%alias` *`alias_name cmd`*	Определить псевдоним команды оболочки
`%bookmark`	Воспользоваться системой закладок IPython
`%cd` *`каталог`*	Сделать указанный каталог рабочим
`%pwd`	Вернуть текущий рабочий каталог
`%pushd` *`каталог`*	Поместить текущий каталог в стек и перейти в указанный каталог
`%popd`	Извлечь каталог из стека и перейти в него
`%dirs`	Вернуть список, содержащий текущее состояние стека каталогов
`%dhist`	Напечатать историю посещения каталогов
`%env`	Вернуть переменные среды в виде словаря

Команды оболочки и псевдонимы

Восклицательный знак `!` в начале командной строки IPython означает, что все следующее за ним следует выполнить в оболочке системы. Таким образом можно удалять файлы (командой `rm` или `del` в зависимости от ОС), изменять рабочий каталог или исполнять другой процесс. Можно даже запустить процесс, который перенимает управление у IPython, даже еще один интерпретатор Python:

```
In [2]: !python
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-44)] on linux2
```

```
Type "packages", "demo" or "enthought" for more information.
>>>
```

Все, что команда выводит на консоль, можно сохранить в переменной, присвоив ей значение выражения, начинающегося со знака `!`. Например, на своей Linux-машине, подключенной к Интернету Ethernet-кабелем, я могу следующим образом записать в переменную Python свой IP-адрес:

```
In [1]: ip_info = !ifconfig eth0 | grep "inet "

In [2]: ip_info[0].strip()
Out[2]: 'inet addr:192.168.1.137 Bcast:192.168.1.255 Mask:255.255.255.0'
```

Возвращенный объект Python `ip_info` – это специализированный список, содержащий различные варианты вывода на консоль.

IPython умеет также подставлять в команды, начинающиеся знаком `!`, значения переменных Python, определенных в текущем окружении. Для этого имени переменной нужно предпослать знак `$`:

```
In [3]: foo = 'test*'
In [4]: !ls $foo
test4.py test.py test.xml
```

Магическая функция `%alias` позволяет определять собственные сокращения для команд оболочки, например:

```
In [1]: %alias ll ls -l

In [2]: ll /usr
total 332
drwxr-xr-x 2 root root 69632 2012-01-29 20:36 bin/
drwxr-xr-x 2 root root 4096 2010-08-23 12:05 games/
drwxr-xr-x 123 root root 20480 2011-12-26 18:08 include/
drwxr-xr-x 265 root root 126976 2012-01-29 20:36 lib/
drwxr-xr-x 44 root root 69632 2011-12-26 18:08 lib32/
lrwxrwxrwx 1 root root 3 2010-08-23 16:02 lib64 -> lib/
drwxr-xr-x 15 root root 4096 2011-10-13 19:03 local/
drwxr-xr-x 2 root root 12288 2012-01-12 09:32 sbin/
drwxr-xr-x 387 root root 12288 2011-11-04 22:53 share/
drwxrwsr-x 24 root src 4096 2011-07-17 18:38 src/
```

Несколько команд можно выполнить, как одну, разделив их точками с запятой:

```
In [558]: %alias test_alias (cd ch08; ls; cd ..)

In [559]: test_alias
macrodata.csv spx.csv tips.csv
```

Обратите внимание, что IPython «забывает» все определенные интерактивно псевдонимы после закрытия сеанса. Чтобы создать постоянные псевдонимы, нужно прибегнуть к системе конфигурирования. Она описывается ниже в этой главе.

Система закладок на каталоги

В IPython имеется простая система закладок, позволяющая создавать псевдонимы часто используемых каталогов, чтобы упростить переход в них. Например, я регулярно захожу в каталог Dropbox, поэтому могу определить закладку, которая даст возможность быстро перейти в него:

```
In [6]: %bookmark db /home/wesm/Dropbox/
```

После этого с помощью магической команды `%cd` я смогу воспользоваться ранее определенными закладками:

```
In [7]: cd db
(bookmark:db) -> /home/wesm/Dropbox/
/home/wesm/Dropbox
```

Если имя закладки конфликтует с именем подкаталога вашего текущего рабочего каталога, то с помощью флага `-b` можно отдать приоритет закладке. Команда `%bookmark` с флагом `-l` выводит список всех закладок:

```
In [8]: %bookmark -l
Current bookmarks:
db -> /home/wesm/Dropbox/
```

Закладки, в отличие от псевдонимов, автоматически сохраняются после закрытия сеанса.

Средства разработки программ

IPython не только является удобной средой для интерактивных вычислений и исследования данных, но и прекрасно оснащен для разработки программ. В приложениях для анализа данных, прежде всего, важно, чтобы код был *правильным*. К счастью, в IPython встроен отлично интегрированный и улучшенный отладчик Python `pdb`. Кроме того, код должен быть *быстрым*. Для этого в IPython имеются простые в использовании средства хронометража и профилирования. Ниже я расскажу об этих инструментах подробнее.

Интерактивный отладчик

Отладчик IPython дополняет `pdb` завершением по нажатию клавиши **Tab**, подсветкой синтаксиса и контекстом для каждой строки трассировки исключения. Отлаживать программу лучше всего сразу после возникновения ошибки. Команда `%debug`, выполненная сразу после исключения, вызывает «посмертный» отладчик и переходит в то место стека вызовов, где было возбуждено исключение:

```
In [2]: run ch03/ipython_bug.py
---------------------------------------------------------------------------
AssertionError Traceback (most recent call last)
/home/wesm/book_scripts/ch03/ipython_bug.py in <module>()
```

```
     13      throws_an_exception()
     14
---> 15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in calling_things()
     11 def calling_things():
     12      works_fine()
---> 13      throws_an_exception()
     14
     15 calling_things()

/home/wesm/book_scripts/ch03/ipython_bug.py in throws_an_exception()
      7      a = 5
      8      b = 6
----> 9      assert(a + b == 10)
     10
     11 def calling_things():

AssertionError:

In [3]: %debug
> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception()
      8      b = 6
----> 9      assert(a + b == 10)
     10
ipdb>
```

Находясь в отладчике, можно выполнять произвольный Python-код и просматривать все объекты и данные (которые интерпретатор «сохранил живыми») в каждом кадре стека. По умолчанию отладчик оказывается на самом нижнем уровне – там, где произошла ошибка. Клавиши `u` (вверх) и `d` (вниз) позволяют переходить с одного уровня стека на другой:

```
ipdb> u
> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()
     12      works_fine()
---> 13      throws_an_exception()
     14
```

Команда `%pdb` устанавливает режим, в котором IPython автоматически вызывает отладчик после любого исключения, многие считают этот режим особенно полезным.

Отладчик также помогает разрабатывать код, особенно когда хочется расставить точки останова либо пройти функцию или скрипт в пошаговом режиме, изучая состояния после каждого шага. Сделать это можно несколькими способами. Первый – воспользоваться функцией `%run` с флагом `-d`, которая вызывает отладчик, перед тем как начать выполнение кода в переданном скрипте. Для входа в скрипт нужно сразу же нажать `s` (step – пошаговый режим):

```
In [5]: run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1
```

```
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> s
--Call--
> /home/wesm/book_scripts/ch03/ipython_bug.py(1)<module>()
1---> 1 def works_fine():
      2     a = 5
      3     b = 6
```

После этого вы сами решаете, каким образом работать с файлом. Например, в приведенном выше примере исключения можно было бы поставить точку останова прямо перед вызовом метода `works_fine` и выполнить программу до этой точки, нажав `c` (continue – продолжить):

```
ipdb> b 12
ipdb> c
> /home/wesm/book_scripts/ch03/ipython_bug.py(12)calling_things()
     11 def calling_things():
2--> 12     works_fine()
     13     throws_an_exception()
```

В этот момент можно войти внутрь `works_fine()` командой `step` или выполнить `works_fine()` без захода внутрь, т. е. перейти к следующей строке, нажав `n` (next – дальше):

```
ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(13)calling_things()
2    12     works_fine()
---> 13     throws_an_exception()
     14
```

Далее мы можем войти внутрь `throws_an_exception`, дойти до строки, где возникает ошибка, и изучить переменные в текущей области видимости. Отметим, что у команд отладчика больший приоритет, чем у имен переменных, поэтому для просмотра переменной с таким же именем, как у команды, необходимо предпослать ей знак `!`.

```
ipdb> s
--Call--
> /home/wesm/book_scripts/ch03/ipython_bug.py(6)throws_an_exception()
      5
----> 6 def throws_an_exception():
      7     a = 5

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(7)throws_an_exception()
      6 def throws_an_exception():
----> 7     a = 5
      8     b = 6

ipdb> n
```

```
> /home/wesm/book_scripts/ch03/ipython_bug.py(8)throws_an_exception()
      7      a = 5
----> 8      b = 6
      9      assert(a + b == 10)

ipdb> n
> /home/wesm/book_scripts/ch03/ipython_bug.py(9)throws_an_exception()
      8      b = 6
----> 9 assert(a + b == 10)
     10

ipdb> !a
5
ipdb> !b
6
```

Уверенное владение интерактивным отладчиком приходит с опытом и практикой. В табл. 3.3 приведен полный перечень команд отладчика. Если вы привыкли к IDE, то консольный отладчик на первых порах может показаться неуклюжим, но со временем это впечатление рассеется. В большинстве IDE для Python имеются отличные графические отладчики, но обычно отладка в самом IPython оказывается намного продуктивнее.

Таблица 3.4. Команда отладчика (I)Python

Команда	Действие
`h(elp)`	Вывести список команд
`help` *команда*	Показать документацию по *команде*
`c(ontinue)`	Продолжить выполнение программы
`q(uit)`	Выйти из отладчика, прекратив выполнение кода
`b(reak)` *номер*	Поставить точку остановка на строке с указанным *номером* в текущем файле
`b` *путь/к/файлу.ру:номер*	Поставить точку остановка на строке с указанным *номером* в указанном файле
`s(tep)`	Войти внутрь функции
`n(ext)`	Выполнить текущую строку и перейти к следующей на текущем уровне
`u(p) / d(own)`	Перемещение вверх и вниз по стеку вызовов
`a(rgs)`	Показать аргументы текущей функции
`debug` *предложение*	Выполнить *предложение* в новом (вложенном) отладчике
`l(ist)` *предложение*	Показать текущую позицию и контекст на текущем уровне стека
`w(here)`	Распечатать весь стек в контексте текущей позиции

Другие способы работы с отладчиком

Существует еще два полезных способа вызова отладчика. Первый – воспользоваться специальной функцией `set_trace` (названной так по аналогии с `pdb.set_trace`), которая по существу является упрощенным вариантом точки останова. Вот два небольших фрагмента, которые вы можете сохранить где-нибудь и использовать в разных программах (я, например, помещаю их в свой профиль IPython):

```
def set_trace():
  from IPython.core.debugger import Pdb
  Pdb(color_scheme='Linux').set_trace(sys._getframe().f_back)

def debug(f, *args, **kwargs):
  from IPython.core.debugger import Pdb
  pdb = Pdb(color_scheme='Linux')
  return pdb.runcall(f, *args, **kwargs)
```

Первая функция, `set_trace`, совсем простая. Вызывайте ее в той точке кода, где хотели бы остановиться и оглядеться (например, прямо перед строкой, в которой происходит исключение):

```
In [7]: run ch03/ipython_bug.py
> /home/wesm/book_scripts/ch03/ipython_bug.py(16)calling_things()
     15      set_trace()
---> 16      throws_an_exception()
     17
```

При нажатии `c` (продолжить) выполнение программы возобновится без каких-либо побочных эффектов. Функция `debug` позволяет вызвать интерактивный отладчик в момент обращения к любой функции. Допустим, мы написали такую функцию:

```
def f(x, y, z=1):
  tmp = x + y
  return tmp / z
```

и хотели бы пройти ее в пошаговом режиме. Обычно `f` используется примерно так: `f(1, 2, z=3)`. А чтобы войти в эту функцию, передайте `f` в качестве первого аргумента функции `debug`, а затем ее позиционные и именованные аргументы:

```
In [6]: debug(f, 1, 2, z=3)
> <ipython-input>(2)f()
      1 def f(x, y, z):
----> 2     tmp = x + y
      3     return tmp / z
ipdb>
```

Мне эти две простенькие функции ежедневно экономят уйму времени. Наконец, отладчик можно использовать в сочетании с функцией `%run`. Запустив скрипт

командой `%run -d`, вы попадете прямо в отладчик и сможете расставить точки останова и начать выполнение:

```
In [1]: %run -d ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:1
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb>
```

Если добавить еще флаг `-b`, указав номер строки, то после входа в отладчик на этой строке уже будет стоять точка останова:

```
In [2]: %run -d -b2 ch03/ipython_bug.py
Breakpoint 1 at /home/wesm/book_scripts/ch03/ipython_bug.py:2
NOTE: Enter 'c' at the ipdb> prompt to start your script.
> <string>(1)<module>()

ipdb> c
> /home/wesm/book_scripts/ch03/ipython_bug.py(2)works_fine()
      1 def works_fine():
1---> 2     a = 5
      3     b = 6
ipdb>
```

Хронометраж программы: %time и %timeit

Для больших или долго работающих аналитических приложений бывает желательно измерить время выполнения различных участков кода или даже отдельных предложений или вызовов функций. Интересно получить отчет о том, какие функции занимают больше всего времени в сложном процессе. По счастью, IPython позволяет без труда получить эту информацию по ходу разработки и тестирования программы.

Ручной хронометраж с помощью встроенного модуля `time` и его функций `time.clock` и `time.time` зачастую оказывается скучной и утомительной процедурой, поскольку приходится писать один и тот же неинтересный код:

```
import time
start = time.time()
for i in range(iterations):
    # здесь код, который требует хронометрировать
elapsed_per = (time.time() - start) / iterations
```

Поскольку эта операция встречается очень часто, в IPython есть две магические функции, `%time` и `%timeit`, которые помогают автоматизировать процесс. Функция `%time` выполняет предложение один раз и сообщает, сколько было затрачено времени. Допустим, имеется длинный список строк, и мы хотим сравнить различные методы выбора всех строк, начинающихся с заданного префикса. Вот простой список, содержащий 700 000 строк, и два метода выборки тех, что начинаются с `'foo'`:

```
# очень длинный список строк
strings = ['foo', 'foobar', 'baz', 'qux',
           'python', 'Guido Van Rossum'] * 100000

method1 = [x for x in strings if x.startswith('foo')]

method2 = [x for x in strings if x[:3] == 'foo']
```

На первый взгляд, производительность должна быть примерно одинаковой, верно? Проверим с помощью функции `%time`:

```
In [561]: %time method1 = [x for x in strings if x.startswith('foo')]
CPU times: user 0.19 s, sys: 0.00 s, total: 0.19 s
Wall time: 0.19 s

In [562]: %time method2 = [x for x in strings if x[:3] == 'foo']
CPU times: user 0.09 s, sys: 0.00 s, total: 0.09 s
Wall time: 0.09 s
```

Наибольший интерес представляет величина `Wall time` (фактическое время). Похоже, первый метод работает в два раза медленнее второго, но это не очень точное измерение. Если вы несколько раз сами замерите время работы этих двух предложений, то убедитесь, что результаты варьируются. Для более точного измерения воспользуемся магической функцией `%timeit`. Она получает произвольное предложение и, применяя внутренние эвристики, выполняет его столько раз, сколько необходимо для получения сравнительно точного среднего времени:

```
In [563]: %timeit [x for x in strings if x.startswith('foo')]
10 loops, best of 3: 159 ms per loop

In [564]: %timeit [x for x in strings if x[:3] == 'foo']
10 loops, best of 3: 59.3 ms per loop
```

Этот, на первый взгляд, безобидный пример показывает, насколько важно хорошо понимать характеристики производительности стандартной библиотеки Python, NumPy, pandas и других используемых в книге библиотек. В больших приложениях для анализа данных из миллисекунд складываются часы!

Функция `%timeit` особенно полезна для анализа предложений и функций, работающих очень быстро, порядка микросекунд (10^{-6} секунд) или наносекунд (10^{-9} секунд). Вроде бы совсем мизерные промежутки времени, но если функцию, работающую 20 микросекунд, вызвать миллион раз, то будет потрачено на 15 секунд больше, чем если бы она работала всего 5 микросекунд. В примере выше можно сравнить две операции со строками напрямую, это даст отчетливое представление об их характеристиках в плане производительности:

```
In [565]: x = 'foobar'

In [566]: y = 'foo'

In [567]: %timeit x.startswith(y)
```

```
1000000 loops, best of 3: 267 ns per loop

In [568]: %timeit x[:3] == y
10000000 loops, best of 3: 147 ns per loop
```

Простейшее профилирование: %prun и %run -p

Профилирование кода тесно связано с хронометражем, только отвечает на вопрос, *где именно* тратится время. В Python основное средство профилирования – модуль `cProfile`, который предназначен отнюдь не только для IPython. `cProfile` исполняет программу или произвольный блок кода и следит за тем, сколько времени проведено в каждой функции.

Обычно `cProfile` запускают из командной строки, профилируют программу целиком и выводят агрегированные временные характеристики каждой функции. Пусть имеется простой скрипт, который выполняет в цикле какой-нибудь алгоритм линейной алгебры (скажем, вычисляет максимальное по абсолютной величине собственное значение для последовательности матриц размерности 100×100):

```
import numpy as np
from numpy.linalg import eigvals

def run_experiment(niter=100):
  K = 100
  results = []
  for _ in xrange(niter):
     mat = np.random.randn(K, K)
     max_eigenvalue = np.abs(eigvals(mat)).max()
     results.append(max_eigenvalue)
  return results
  some_results = run_experiment()
  print 'Самое большое встретившееся: %s' % np.max(some_results)
```

Незнание NumPy пусть вас не пугает. Это скрипт можно запустить под управлением `cProfile` из командной строки следующим образом:

```
python -m cProfile cprof_example.py
```

Попробуйте и убедитесь, что результаты отсортированы по имени функции. Такой отчет не позволяет сразу увидеть, где тратится время, поэтому обычно *порядок сортировки* задают с помощью флага `-s`:

```
$ python -m cProfile -s cumulative cprof_example.py
Самое большое встретившееся: 11.923204422
15116 function calls (14927 primitive calls) in 0.720 seconds
Ordered by: cumulative time
ncalls  tottime percall  cumtime percall  filename:lineno(function)
     1    0.001   0.001    0.721   0.721  cprof_example.py:1(<module>)
   100    0.003   0.000    0.586   0.006  linalg.py:702(eigvals)
   200    0.572   0.003    0.572   0.003  {numpy.linalg.lapack_lite.dgeev}
     1    0.002   0.002    0.075   0.075  __init__.py:106(<module>)
```

```
   100    0.059    0.001    0.059    0.001  {method 'randn')
     1    0.000    0.000    0.044    0.044  add_newdocs.py:9(<module>)
     2    0.001    0.001    0.037    0.019  __init__.py:1(<module>)
     2    0.003    0.002    0.030    0.015  __init__.py:2(<module>)
     1    0.000    0.000    0.030    0.030  type_check.py:3(<module>)
     1    0.001    0.001    0.021    0.021  __init__.py:15(<module>)
     1    0.013    0.013    0.013    0.013  numeric.py:1(<module>)
     1    0.000    0.000    0.009    0.009  __init__.py:6(<module>)
     1    0.001    0.001    0.008    0.008  __init__.py:45(<module>)
   262    0.005    0.000    0.007    0.000  function_base.py:3178(add_newdoc)
   100    0.003    0.000    0.005    0.000  linalg.py:162(_assertFinite)
   ...
```

Показаны только первые 15 строк отчета. Читать его проще всего, просматривая сверху вниз столбец `cumtime`, чтобы понять, сколько времени было проведено *внутри* каждой функции. Отметим, что если одна функция вызывает другую, то *таймер не останавливается*. `cProfile` запоминает моменты начала и конца каждого вызова функции и на основе этих данных создает отчет о затраченном времени.

`cProfile` можно запускать не только из командной строки, но и программно для профилирования работы произвольных блоков кода без порождения нового процесса. В IPython имеется удобный интерфейс к этой функциональности в виде команды `%prun` и команды `%run` с флагом `-p`. Команда `%prun` принимает те же «аргументы командной строки», что и `cProfile`, но профилирует произвольное предложение Python, а не `py`-файл:

```
In [4]: %prun -l 7 -s cumulative run_experiment()
        4203 function calls in 0.643 seconds

Ordered by: cumulative time
List reduced from 32 to 7 due to restriction <7>

ncalls  tottime percall cumtime  percall  filename:lineno(function)
     1    0.000   0.000   0.643    0.643  <string>:1(<module>)
     1    0.001   0.001   0.643    0.643  cprof_example.py:4(run_experiment)
   100    0.003   0.000   0.583    0.006  linalg.py:702(eigvals)
   200    0.569   0.003   0.569    0.003  {numpy.linalg.lapack_lite.dgeev}
   100    0.058   0.001   0.058    0.001  {method 'randn'}
   100    0.003   0.000   0.005    0.000  linalg.py:162(_assertFinite)
   200    0.002   0.000   0.002    0.000  {method 'all' of 'numpy.ndarray' objects}
```

Аналогично команда `%run -p -s cumulative cprof_example.py` дает тот же результат, что рассмотренный выше запуск из командной строки, только не приходится выходить из IPython.

Построчное профилирование функции

Иногда информации, полученной от `%prun` (или добытой иным способом профилирования на основе `cProfile`), недостаточно, чтобы составить полное представление о времени работы функции. Или она настолько сложна, что результаты, агрегированные по имени функции, с трудом поддаются интерпретации. На такой

случай есть небольшая библиотека `line_profiler` (ее поможет установить PyPI или любой другой инструмент управления пакетами). Она содержит расширение IPython, включающее новую магическую функцию `%lprun`, которая строит построчный профиль выполнения одной или нескольких функций. Чтобы подключить это расширение, нужно модифицировать конфигурационный файл IPython (см. документацию по IPython или раздел, посвященный конфигурированию, ниже), добавив такую строку:

```
# Список имен загружаемых модулей с расширениями IPython.
c.TerminalIPythonApp.extensions = ['line_profiler']
```

Библиотеку `line_profiler` можно использовать из программы (см. полную документацию), но, пожалуй, наиболее эффективна интерактивная работа с ней в IPython. Допустим, имеется модуль `prof_mod`, содержащий следующий код, в котором выполняются операции с массивом NumPy:

```
from numpy.random import randn

def add_and_sum(x, y):
  added = x + y
  summed = added.sum(axis=1)
  return summed

def call_function():
  x = randn(1000, 1000)
  y = randn(1000, 1000)
  return add_and_sum(x, y)
```

Если бы нам нужно было оценить производительность функции `add_and_sum`, то команда `%prun` дала бы такие результаты:

```
In [569]: %run prof_mod

In [570]: x = randn(3000, 3000)

In [571]: y = randn(3000, 3000)

In [572]: %prun add_and_sum(x, y)
          4 function calls in 0.049 seconds
   Ordered by: internal time
   ncalls tottime percall cumtime percall filename:lineno(function)
       1   0.036   0.036   0.046   0.046 prof_mod.py:3(add_and_sum)
       1   0.009   0.009   0.009   0.009 {method 'sum' of 'numpy.ndarray' objects}
       1   0.003   0.003   0.049   0.049 <string>:1(<module>)
       1   0.000   0.000   0.000   0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Не слишком полезно. Но после активации расширения IPython `line_profiler` становится доступна новая команда `%lprun`. От `%prun` она отличается только тем, что мы указываем, какую функцию (или функции) хотим профилировать. Порядок вызова такой:

```
%lprun -f func1 -f func2 профилируемое_предложение
```

В данном случае мы хотим профилировать функцию `add_and_sum`, поэтому пишем:

```
In [573]: %lprun -f add_and_sum add_and_sum(x, y)
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.045936 s
Line     # Hits     Time    Per Hit    % Time  Line Contents
==============================================================
   3                                           def add_and_sum(x, y):
   4          1   36510    36510.0     79.5       added = x + y
   5          1    9425     9425.0     20.5       summed = added.sum(axis=1)
   6          1       1        1.0      0.0       return summed
```

Согласитесь, так гораздо понятнее. В этом примере мы профилировали ту же функцию, которая составляла предложение. Но можно было бы вызвать функцию `call_function` из показанного выше модуля и профилировать ее наряду с `add_and_sum`, это дало бы полную картину производительности кода:

```
In [574]: %lprun -f add_and_sum -f call_function call_function()
Timer unit: 1e-06 s
File: book_scripts/prof_mod.py
Function: add_and_sum at line 3
Total time: 0.005526 s
Line     # Hits     Time    Per Hit    % Time  Line Contents
==============================================================
   3                                           def add_and_sum(x, y):
   4          1     4375     4375.0     79.2       added = x + y
   5          1     1149     1149.0     20.8       summed = added.sum(axis=1)
   6          1        2        2.0      0.0       return summed
File: book_scripts/prof_mod.py
Function: call_function at line 8
Total time: 0.121016 s
Line     # Hits     Time    Per Hit    % Time  Line Contents
==============================================================
   8                                           def call_function():
   9          1    57169    57169.0     47.2       x = randn(1000, 1000)
  10          1    58304    58304.0     48.2       y = randn(1000, 1000)
  11          1     5543     5543.0      4.6       return add_and_sum(x, y)
```

Обычно я предпочитаю использовать `%prun` (`cProfile`) для «макропрофилирования», а `%lprun` (`line_profiler`) – для «микропрофилирования». Полезно освоить оба инструмента.

Явно указывать имена подлежащих профилированию функций в команде `%lprun` необходимо, потому что накладные расходы на «трассировку» времени выполнения каждой строки весьма значительны. Трассировка функций, не представляющих интереса, может существенно изменить результаты профилирования.

HTML-блокнот в IPython

В 2011 году команда разработчиков IPython, возглавляемая Брайаном Грейнджером, приступила к разработке основанного на веб-технологиях формата интерактивного вычислительного документа под названием блокнот IPython (IPython Notebook). Со временем он превратился в чудесный инструмент для интерактивных вычислений и идеальное средство для воспроизводимых исследований и преподавания. Я пользовался им при написании большинства примеров для этой книги, призываю и вас не пренебрегать им.

Формат `ipynb`-документа основан на JSON и позволяет легко обмениваться кодом, результатами и рисунками. На недавних конференциях по Python получил широкое распространение такой подход к демонстрациям: создать `ipynb`-файлы в блокноте и разослать их всем желающим для экспериментов.

Блокнот реализован в виде облегченного серверного процесса, запускаемого из командной строки, например:

```
$ ipython notebook --pylab=inline
[NotebookApp] Using existing profile dir: u'/home/wesm/.config/ipython/profile_default'
[NotebookApp] Serving notebooks from /home/wesm/book_scripts
[NotebookApp] The IPython Notebook is running at: http://127.0.0.1:8888/
[NotebookApp] Use Control-C to stop this server and shut down all kernels.
```

На большинстве платформ автоматически откроется браузер, подразумеваемый по умолчанию, и в нем появится информационная панель блокнота (рис. 3.4). Иногда приходится переходить на нужный URL-адрес самостоятельно. После этого можно создать новый блокнот и приступить к исследованиям.

Поскольку блокнот работает внутри браузера, серверный процесс можно запустить где угодно. Можно даже организовать безопасное соединение с блокнотами, работающими в облаке, например Amazon EC2. На момент написания этой книги уже существовал новый проект NotebookCloud (*http://notebookcloud.appspot.com*), который позволяет без труда запускать блокноты в EC2.

Советы по продуктивной разработке кода с использованием IPython

Создание кода таким образом, чтобы его можно было разрабатывать, отлаживать и в конечном счете *использовать* интерактивно, многим может показаться сменой парадигмы. Придется несколько изменить подходы к таким процедурным деталям, как перезагрузка кода, а также сам стиль кодирования.

Поэтому изложенное в этом разделе – скорее искусство, чем наука, вы должны будете экспериментально определить наиболее эффективный для себя способ написания Python-кода. Конечная задача – структурировать код так, чтобы с ним было легко работать интерактивно и изучать результаты прогона всей программы или отдельной функции с наименьшими усилиями. Я пришел к выводу, что программу, спроектированную в расчете на IPython, использовать проще, чем анало-

гичную, но построенную как автономное командное приложение. Это становится особенно важно, когда возникает какая-то проблема и нужно найти ошибку в коде, написанном вами или кем-то еще несколько месяцев или лет назад.

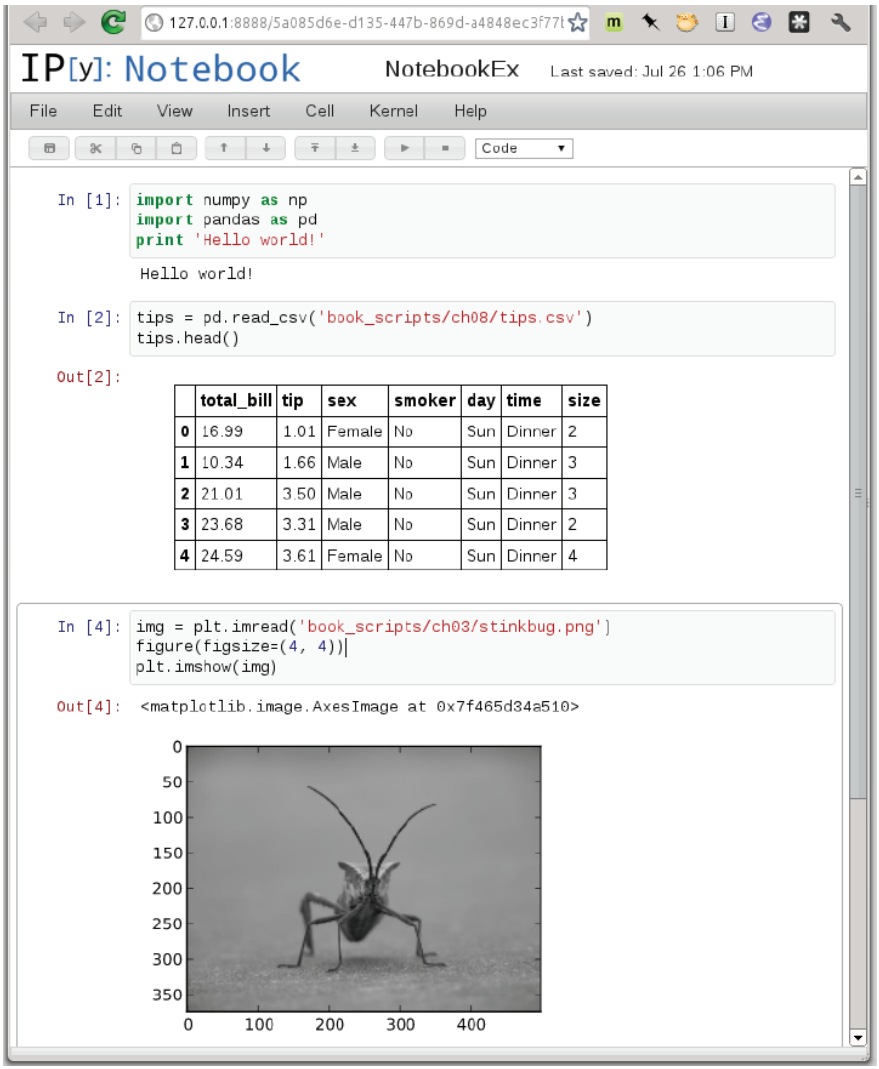


Рис. 3.4. Блокнот IPython

Перезагрузка зависимостей модуля

Когда в Python-программе впервые встречается предложение `import some_lib`, выполняется код из модуля `some_lib` и все переменные, функции и импортированные модули сохраняются во вновь созданном пространстве имен модуля

some_lib. При следующей обработке предложения import some_lib будет возвращена ссылка на уже существующее пространство имен модуля. При интерактивной разработке кода возникает проблема: как быть, когда, скажем, с помощью команды %run выполняется скрипт, зависящий от другого модуля, в который вы внесли изменения? Допустим, в файле test_script.py находится такой код:

```
import some_lib

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

Если выполнить %run test_script.py, а затем изменить some_lib.py, то при следующем выполнении %run test_script.py мы получим *старую версию* some_lib из-за принятого в Python механизма однократной загрузки. Такое поведение отличается от некоторых других сред анализа данных, например MATLAB, в которых изменения кода распространяются автоматически[2]. Справиться с этой проблемой можно двумя способами. Во-первых, использовать встроенную в Python функцию reload, изменив test_script.py следующим образом:

```
import some_lib
reload(some_lib)

x = 5
y = [1, 2, 3, 4]
result = some_lib.get_answer(x, y)
```

При этом гарантируется получение новой копии some_lib при каждом запуске test_script.py. Очевидно, что если глубина вложенности зависимостей больше единицы, то вставлять reload повсюду становится утомительно. Поэтому в IPython имеется специальная функция dreload (*не* магическая), выполняющая «глубокую» (рекурсивную) перезагрузку модулей. Если бы я написал import some_lib, а затем dreload(some_lib), то был бы перезагружен как модуль some_lib, так и все его зависимости. К сожалению, это работает не во всех случаях, но если работает, то оказывается куда лучше перезапуска всего IPython.

Советы по проектированию программ

Простых рецептов здесь нет, но некоторыми общими соображениями, которые лично мне кажутся эффективными, я все же поделюсь.

Сохраняйте ссылки на нужные объекты и данные

Программы, рассчитанные на запуск из командной строки, нередко структурируются, как показано в следующем тривиальном примере:

[2] Поскольку модуль или пакет может импортироваться в нескольких местах программы, Python кэширует код модуля при первом импортировании, а не выполняет его каждый раз. В противном случае следование принципам модульности и правильной организации кода могло бы поставить под угрозу эффективность приложения.

```
from my_functions import g

def f(x, y):
  return g(x + y)

def main():
  x = 6
  y = 7.5
  result = x + y

if __name__ == '__main__':
  main()
```

Вы уже видите, что случится, если эту программу запустить в IPython? После ее завершения все результаты или объекты, определенные в функции `main`, будут недоступны в оболочке IPython. Лучше, если любой код, находящийся в `main`, будет исполняться прямо в глобальном пространстве имен модуля (или в блоке `if __name__ == '__main__':`, если вы хотите, чтобы и сам модуль был импортируемым). Тогда после выполнения кода командой `%run` вы сможете просмотреть все переменные, определенные в `main`. В таком простом примере это неважно, но далее в книге будут рассмотрены сложные задачи анализа данных, в которых участвуют большие наборы, и их исследование может оказаться весьма полезным.

Плоское лучше вложенного

Глубоко вложенный код напоминает мне чешуи луковицы. Сколько чешуй придется снять при тестировании или отладке функции, чтобы добраться до интересующего кода? Идея «плоское лучше вложенного» – часть «Свода мудрости Python», применимая и к разработке кода, предназначенного для интерактивного использования. Чем более модульными являются классы и функции и чем меньше связей между ними, тем проще их тестировать (если вы пишете автономные тесты), отлаживать и использовать интерактивно.

Перестаньте бояться длинных файлов

Если вы раньше работали с Java (или аналогичным языком), то, наверное, вам говорили, что чем файл короче, тем лучше. Во многих языках это разумный совет; длинный файл несет в себе дурной «запашок» и наводит на мысль о необходимости рефакторинга или реорганизации. Однако при разработке кода в IPython наличие 10 мелких (скажем, не более 100 строчек) взаимосвязанных файлов с большей вероятностью вызовет проблемы, чем при работе всего с одним, двумя или тремя файлами подлиннее. Чем меньше файлов, тем меньше нужно перезагружать модулей и тем реже приходится переходить от файла к файлу в процессе редактирования. Я пришел к выводу, что сопровождение крупных модулей с высокой степенью *внутренней* сцепленности гораздо полезнее и лучше соответствует духу Python. По мере приближения к окончательному решению, возможно, имеет смысл разбить большие файлы на более мелкие.

Понятно, что я не призываю бросаться из одной крайности в другую, т. е. помещать весь код в один гигантский файл. Для отыскания разумной и интуитив-

но очевидной структуры модулей и пакетов, составляющих большую программу, нередко приходится потрудиться, но при коллективной работе это очень важно. Каждый модуль должен обладать внутренней сцепленностью, а местонахождение функций и классов, относящихся к каждой области функциональности, должно быть как можно более очевидным.

Дополнительные возможности IPython

Делайте классы дружественными к IPython

В IPython предпринимаются все меры к тому, чтобы вывести на консоль понятное строковое представление инспектируемых объектов. Для многих объектов, в частности словарей, списков и кортежей, красивое форматирование обеспечивается за счет встроенного модуля `pprint`. Однако в классах, определенных пользователем, порождение строкового представления возлагается на автора. Рассмотрим такой простенький класс:

```
class Message:
  def __init__(self, msg):
     self.msg = msg
```

Вы будете разочарованы тем, как такой класс распечатывается по умолчанию:

```
In [576]: x = Message('I have a secret')

In [577]: x
Out[577]: <__main__.Message instance at 0x60ebbd8>
```

IPython принимает строку, возвращенную магическим методом `__repr__` (выполняя предложение `output = repr(obj)`), и выводит ее на консоль. Но раз так, то мы можем включить в класс простой метод `__repr__`, который создает более полезное представление:

```
class Message:

  def __init__(self, msg):
     self.msg = msg

  def __repr__(self):
     return 'Message: %s' % self.msg

In [579]: x = Message('У меня есть секрет')

In [580]: x
Out[580]: Message: У меня есть секрет
```

Профили и конфигурирование

Многие аспекты внешнего вида (цвета, приглашение, расстояние между строками и т. д.) и поведения оболочки IPython настраиваются с помощью развитой

системы конфигурирования. Приведем лишь несколько примеров того, что можно сделать.

- Изменить цветовую схему.
- Изменить вид приглашений ввода и вывода или убрать пустую строку, печатаемую после `Out` и перед следующим приглашением `In`.
- Выполнить список произвольных предложений Python. Это может быть, например, импорт постоянно используемых модулей или вообще все, что должно выполняться сразу после запуска IPython.
- Включить расширения IPython, например магическую функцию `%lprun` в модуле `line_profiler`.
- Определить собственные магические функции или псевдонимы системных.

Все эти параметры задаются в конфигурационном файле `ipython_config.py`, находящемся в каталоге `~/.config/ipython/` в UNIX-системе или в каталоге `%HOME%/.ipython/` в Windows. Где находится ваш домашний каталог, зависит от системы. Конфигурирование производится на основе конкретного *профиля*. При обычном запуске IPython загружается *профиль по умолчанию*, который хранится в каталоге `profile_default`. Следовательно, в моей Linux-системе полный путь к конфигурационному файлу IPython по умолчанию будет таким:

```
/home/wesm/.config/ipython/profile_default/ipython_config.py
```

Не стану останавливаться на технических деталях содержимого этого файла. По счастью, все параметры в нем подробно прокомментированы, так что оставляю их изучение и изменение читателю. Еще одна полезная возможность – поддержка сразу *нескольких профилей*. Допустим, имеется альтернативная конфигурация IPython для конкретного приложения или проекта. Чтобы создать новый профиль, нужно всего лишь ввести такую строку:

```
ipython profile create secret_project
```

Затем отредактируйте конфигурационные файлы во вновь созданном каталоге `profile_secret_project` и запустите IPython следующим образом:

```
$ ipython --profile=secret_project
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.13 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

IPython profile: secret_project

In [1]:
```

Как всегда, дополнительные сведения о профилях и конфигурировании можно найти в документации по IPython в сети.

Благодарности

Материалы этой главы частично были заимствованы из великолепной документации, подготовленной разработчиками IPython. Я испытываю к ним бесконечную благодарность за создание этого восхитительного набора инструментов.

ГЛАВА 4. Основы NumPy: массивы и векторные вычисления

Numerical Python, или сокращенно NumPy – краеугольный пакет для высокопроизводительных научных расчетов и анализа данных. Это фундамент, на котором возведены почти все описываемые в этой книге высокоуровневые инструменты. Вот лишь часть того, что предлагается.

- `ndarray`, быстрый и потребляющий мало памяти многомерный массив, предоставляющий векторные арифметические операции и возможность *укладывания*.
- Стандартные математические функции для выполнения быстрых операций над целыми массивами без явного выписывания циклов.
- Средства для чтения массива данных с диска и записи его на диск, а также для работы с проецируемыми на память файлами.
- Алгоритмы линейной алгебры, генерация случайных чисел и преобразование Фурье.
- Средства для интеграции с кодом, написанным на C, C++ или Fortran

Последний пункт – один из самых важных с точки зрения экосистемы. Благодаря наличию простого C API в NumPy очень легко передавать данные внешним библиотекам, написанным на языке низкого уровня, а также получать от внешних библиотек данные в виде массивов NumPy. Эта возможность сделала Python излюбленным языком для обертывания имеющегося кода на C/C++/Fortran с приданием ему динамического и простого в использовании интерфейса.

Хотя сам по себе пакет NumPy почти не содержит средств высокоуровневого анализа данных, понимание массивов NumPy и ориентированных на эти массивы вычислений поможет гораздо эффективнее использовать инструменты типа pandas. Если вы только начинаете изучать Python и просто хотите познакомиться с тем, как pandas позволяет работать с данными, можете лишь бегло просмотреть эту главу. Более сложные средства NumPy, в частности укладывание, рассматриваются в главе 12.

В большинстве приложений для анализа данных основной интерес представляет следующая функциональность:

- быстрые векторные операции для переформатирования и очистки данных, выборки подмножеств и фильтрации, преобразований и других видов вычислений;
- стандартные алгоритмы работы с массивами, например: фильтрация, удаление дубликатов и теоретико-множественные операции;
- эффективная описательная статистика, агрегирование и обобщение данных;
- выравнивание данных и реляционные операции объединения и соединения разнородных наборов данных;
- описание условной логики в виде выражений-массивов вместо циклов с ветвлением `if-elif-else`;
- групповые операции с данными (агрегирование, преобразование, применение функции). Подробнее об этом см. главу 5.

Хотя в NumPy имеются вычислительные основы для этих операций, по большей части для анализа данных (особенно структурированных или табличных) лучше использовать библиотеку pandas, потому что она предлагает развитый высокоуровневый интерфейс для решения большинства типичных задач обработки данных – простой и лаконичный. Кроме того, в pandas есть кое-какая предметно-ориентированная функциональность, например операции с временными рядами, отсутствующая в NumPy.

И в этой главе, и далее в книге я использую стандартное принятое в NumPy соглашение – всегда включать предложение `import numpy as np`. Конечно, никто не мешает добавить в программу предложение `from numpy import *`, чтобы не писать всюду `np.`, но это дурная привычка, от которой я хотел бы вас предостеречь.

NumPy ndarray: объект многомерного массива

Одна из ключевых особенностей NumPy – объект `ndarray` для представления N-мерного массива; это быстрый и гибкий контейнер для хранения больших наборов данных в Python. Массивы позволяют выполнять математические операции над целыми блоками данных, применяя такой же синтаксис, как для соответствующих операций над скалярами:

```
In [8]: data

Out[8]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639, 0.2379, 0.9104]])

In [9]: data * 10                          In [10]: data + data
Out[9]:                                    Out[10]:
```

```
array([[ 9.5256, -2.4601, -8.8565],     array([[ 1.9051, -0.492 , -1.7713],
       [ 5.6385, 2.3794, 9.104 ]])              [ 1.1277, 0.4759, 1.8208]])
```

ndarray – это обобщенный многомерный контейнер для однородных данных, т. е. в нем могут храниться только элементы одного типа. У любого массива есть атрибут `shape` – кортеж, описывающий размер по каждому измерению, и атрибут `dtype` – объект, описывающий *тип данных* в массиве:

```
In [11]: data.shape
Out[11]: (2, 3)

In [12]: data.dtype
Out[12]: dtype('float64')
```

В этой главе вы познакомитесь с основами работы с массивами NumPy в объеме, достаточном для чтения книги. Для многих аналитических приложений глубокое понимание NumPy необязательно, но овладение стилем мышления и методами программирования, ориентированными на массивы, – ключевой этап на пути становления эксперта по применению Python в научных приложениях.

Слова «массив», «массив NumPy» и «ndarray» в этой книге почти всегда означают одно и то же: объект ndarray.

Создание ndarray

Проще всего создать массив с помощью функции `array`. Она принимает любой объект, похожий на последовательность (в том числе другой массив), и порождает новый массив NumPy, содержащий переданные данные. Например, такое преобразование можно проделать со списком:

```
In [13]: data1 = [6, 7.5, 8, 0, 1]
In [14]: arr1 = np.array(data1)

In [15]: arr1
Out[15]: array([ 6. , 7.5, 8. , 0. , 1. ])
```

Вложенные последовательности, например список списков одинаковой длины, можно преобразовать в многомерный массив:

```
In [16]: data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

In [17]: arr2 = np.array(data2)

In [18]: arr2
Out[18]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])

In [19]: arr2.ndim
```

```
Out[19]: 2

In [20]: arr2.shape
Out[20]: (2, 4)
```

Если не определено явно (подробнее об этом ниже), то функция `np.array` пытается самостоятельно определить подходящий тип данных для создаваемого массива. Этот тип данных хранится в специальном объекте `dtype`; например, в примерах выше имеем:

```
In [21]: arr1.dtype
Out[21]: dtype('float64')

In [22]: arr2.dtype
Out[22]: dtype('int64')
```

Помимо `np.array`, существует еще ряд функций для создания массивов. Например, `zeros` и `ones` создают массивы заданной длины, состоящие из нулей и единиц соответственно, а `shape.empty` создает массив, не инициализируя его элементы. Для создания многомерных массивов нужно передать кортеж, описывающий форму:

```
In [23]: np.zeros(10)
Out[23]: array([ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [24]: np.zeros((3, 6))
Out[24]:
array([[ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.],
       [ 0., 0., 0., 0., 0., 0.]])

In [25]: np.empty((2, 3, 2))
Out[25]:
array([[[  4.94065646e-324,   4.94065646e-324],
        [  3.87491056e-297,   2.46845796e-130],
        [  4.94065646e-324,   4.94065646e-324]],
       [[  1.90723115e+083,   5.73293533e-053],
        [ -2.33568637e+124,  -6.70608105e-012],
        [  4.42786966e+160,   1.27100354e+025]]])
```

Предполагать, что `np.empty` возвращает массив из одних нулей, небезопасно. Часто возвращается массив, содержащий неинициализированный мусор, – как в примере выше.

Функция `arange` – вариант встроенной в Python функции `range`, только возвращаемым значением является массив:

```
In [26]: np.arange(15)
Out[26]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

В табл. 4.1 приведен краткий список стандартных функций создания массива. Поскольку NumPy ориентирован, прежде всего, на численные расчеты, тип данных, если он не указан явно, во многих случаях предполагается `float64` (числа с плавающей точкой).

Таблица 4.1. Функции создания массива

Функция	Описание
`array`	Преобразует входные данные (список, кортеж, массив или любую другую последовательность) в ndarray. Тип dtype задается явно или выводится неявно. Входные данные по умолчанию копируются
`asarray`	Преобразует входные данные в ndarray, но не копирует, если на вход уже подан ndarray
`arange`	Аналогична встроенной функции range, но возвращает массив, а не список
`ones, ones_like`	Порождает массив, состоящий из одних единиц, с заданными атрибутами shape и dtype. Функция `ones_like` принимает другой массив и порождает массив из одних единиц с такими же значениями shape и dtype
`zeros, zeros_like`	Аналогичны `ones` и `ones_like`, только порождаемый массив состоит из одних нулей
`empty, empty_like`	Создают новые массивы, выделяя под них память, но, в отличие от `ones` и `zeros`, не инициализируют ее
`eye, identity`	Создают единичную квадратную матрицу N × N (элементы на главной диагонали равны 1, все остальные – 0)

Тип данных для ndarray

Тип данных, или `dtype` – это специальный объект, который содержит информацию, необходимую ndarray для интерпретации содержимого блока памяти:

```
In [27]: arr1 = np.array([1, 2, 3], dtype=np.float64)

In [28]: arr2 = np.array([1, 2, 3], dtype=np.int32)

In [29]: arr1.dtype In [30]: arr2.dtype

Out[29]: dtype('float64') Out[30]: dtype('int32')
```

Объектам `dtype` NumPy в значительной мере обязан своей эффективностью и гибкостью. В большинстве случаев они точно соответствуют внутреннему машинному представлению, что позволяет без труда читать и записывает двоичные потоки данных на диск, а также обмениваться данными с кодом, написанным на языке низкого уровня типа C или Fortran. Числовые dtype именуются единообразно: имя типа, например `float` или `int`, затем число, указывающее разрядность одного элемента. Стандартное значение с плавающей точкой двойной точности (хранящееся во внутреннем представлении объекта Python типа `float`) занимает 8 байтов или

64 бита. Поэтому соответствующий тип в NumPy называется `float64`. В табл. 4.2 приведен полный список поддерживаемых NumPy типов данных.

Не пытайтесь сразу запомнить все типы данных NumPy, особенно если вы только приступаете к изучению. Часто нужно заботиться только об общем *виде* обрабатываемых данных, например: числа с плавающей точкой, комплексные, целые, булевы значения, строки или общие объекты Python. Если необходим более точный контроль над тем, как данные хранятся в памяти или на диске, особенно когда речь идет о больших наборах данных, то знать о возможности такого контроля полезно.

Таблица 4.2. Типы данных NumPy

Функция	Код типа	Описание
`int8, uint8`	`i1, u1`	Знаковое и беззнаковое 8-разрядное (1 байт) целое
`int16, uint16`	`i2, u2`	Знаковое и беззнаковое 16-разрядное (2 байта) целое
`int32, uint32`	`i4, u4`	Знаковое и беззнаковое 32-разрядное (4 байта) целое
`int64, uint64`	`i8, u8`	Знаковое и беззнаковое 64-разрядное (8 байт) целое
`float16`	`f2`	С плавающей точкой половинной точности
`float32`	`f4`	Стандартный тип с плавающей точкой одинарной точности. Совместим с типом C `float`
`float64`	`f8 или d`	Стандартный тип с плавающей точкой двойной точности. Совместим с типом C `double` и с типом Python `float`
`float128`	`f16`	С плавающей точкой расширенной точности
`complex64, complex128, complex256`	`c8, c16, c32`	Комплексные числа, вещественная и мнимая части которых представлены соответственно типами `float32`, `float64` и `float128`
`bool`	`?`	Булев тип, способный хранить значения `True` и `False`
`object`	`O`	Тип объекта Python
`string_`	`S`	Тип строки фиксированной длины (1 байт на символ). Например, строка длиной 10 имеет тип `S10`.
`unicode_`	`U`	Тип Unicode-строки фиксированной длины (количество байтов на символ зависит от платформы). Семантика такая же, как у типа `string_` (например, `U10`).

Можно явно преобразовать, или *привести* массив одного типа к другому, воспользовавшись методом `astype`:

```
In [31]: arr = np.array([1, 2, 3, 4, 5])

In [32]: arr.dtype
Out[32]: dtype('int64')

In [33]: float_arr = arr.astype(np.float64)

In [34]: float_arr.dtype
Out[34]: dtype('float64')
```

Здесь целые были приведены к типу с плавающей точкой. Если бы я попытался привести числа с плавающей точкой к целому типу, то дробная часть была бы отброшена:

```
In [35]: arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [36]: arr
Out[36]: array([ 3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

In [37]: arr.astype(np.int32)
Out[37]: array([ 3, -1, -2, 0, 12, 10], dtype=int32)
```

Если имеется массив строк, представляющих целые числа, то `astype` позволит преобразовать их в числовую форму:

```
In [38]: numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

In [39]: numeric_strings.astype(float)
Out[39]: array([ 1.25, -9.6 , 42. ])
```

Если по какой-то причине выполнить приведение не удастся (например, если строку нельзя преобразовать в тип `float64`), то будет возбуждено исключение `TypeError`. Обратите внимание, что в примере выше я поленился и написал `float` вместо `np.float64`, но NumPy оказался достаточно «умным» – он умеет подменять типы Python эквивалентными dtype.

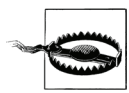
Можно также использовать атрибут dtype другого массива:

```
In [40]: int_array = np.arange(10)

In [41]: calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

In [42]: int_array.astype(calibers.dtype)
Out[42]: array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

На dtype можно сослаться также с помощью коротких кодов типа:

```
In [43]: empty_uint32 = np.empty(8, dtype='u4')

In [44]: empty_uint32
Out[44]:
array([       0,     0,      65904672,       0,      64856792,       0,
       39438163,     0], dtype=uint32)
```

При вызове `astype` *всегда* создается новый массив (данные копируются), даже если новый dtype не отличается от старого.



Следует иметь в виду, что числа с плавающей точкой, например типа `float64` или `float32`, предоставляют дробные величины приближенно. В сложных вычислениях могут накапливаться *ошибки округления*, из-за которых сравнение возможно только с точностью до определенного числа десятичных знаков.

Операции между массивами и скалярами

Массивы важны, потому что позволяют выразить операции над совокупностями данных без выписывания циклов `for`. Обычно это называется *векторизацией*. Любая арифметическая операция над массивами одинакового размера применяется к соответственным элементам:

```
In [45]: arr = np.array([[1., 2., 3.], [4., 5., 6.]])

In [46]: arr
Out[46]:
array([[ 1., 2., 3.],
       [ 4., 5., 6.]])

In [47]: arr * arr
Out[47]:
array([[  1.,  4.,  9.],
       [ 16., 25., 36.]])

In [48]: arr - arr
Out[48]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

Как легко догадаться, арифметические операции, в которых участвует скаляр, применяются к каждому элементу массива:

```
In [49]: 1 / arr
Out[49]:
array([[   1. , 0.5 , 0.3333],
       [ 0.25 , 0.2 , 0.1667]])

In [50]: arr ** 0.5
Out[50]:
array([[ 1. ,  1.4142,  1.7321],
       [ 2. ,  2.2361,  2.4495]])
```

Операции между массивами разного размера называются *укладыванием*, мы будем подробно рассматривать их в главе 12. Глубокое понимание укладывания необязательно для чтения большей части этой книги.

Индексирование и вырезание

Индексирование массивов NumPy – обширная тема, поскольку подмножество массива или его отдельные элементы можно выбрать различными способами. С одномерными массивами все просто; на поверхностный взгляд, они ведут себя, как списки Python:

```
In [51]: arr = np.arange(10)

In [52]: arr
Out[52]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [53]: arr[5]
Out[53]: 5

In [54]: arr[5:8]
Out[54]: array([5, 6, 7])

In [55]: arr[5:8] = 12

In [56]: arr
```

```
Out[56]: array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

Как видите, если присвоить скалярное значение срезу, как в `arr[5:8] = 12`, то оно распространяется (или *укладывается*) на весь срез. Важнейшее отличие от списков состоит в том, что срез массива является *представлением* исходного массива. Это означает, что данные на самом деле не копируются, а любые изменения, внесенные в представление, попадают и в исходный массив.

```
In [57]: arr_slice = arr[5:8]

In [58]: arr_slice[1] = 12345

In [59]: arr
Out[59]: array([    0,     1,     2,     3,     4,    12, 12345,    12,
                8,     9])

In [60]: arr_slice[:] = 64

In [61]: arr
Out[61]: array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

При первом знакомстве с NumPy это может стать неожиданностью, особенно если вы привыкли к программированию массивов в других языках, где копирование данных применяется чаще. Но NumPy проектировался для работы с большими массивами данных, поэтому при безудержном копировании данных неизбежно возникли бы проблемы с быстродействием и памятью.

Чтобы получить копию, а не представление среза массива, нужно выполнить операцию копирования явно, например: `arr[5:8].copy()`.

Для массивов большей размерности и вариантов тоже больше. В случае двумерного массива результатом индексирования с одним индексом является не скаляр, а одномерный массив:

```
In [62]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [63]: arr2d[2]
Out[63]: array([7, 8, 9])
```

К отдельным элементам можно обращаться рекурсивно. Но это слишком громоздко, поэтому для выбора одного элемента можно указать список индексов через запятую. Таким образом, следующие две конструкции эквивалентны:

```
In [64]: arr2d[0][2]
Out[64]: 3

In [65]: arr2d[0, 2]
Out[65]: 3
```

Рисунок 4.1 иллюстрирует индексирование двумерного массива.

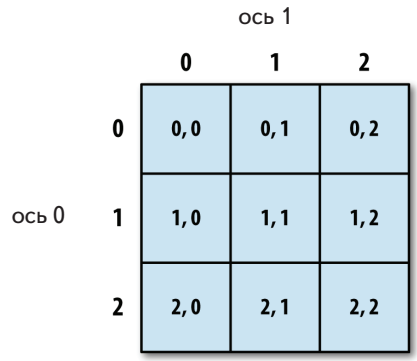


Рис. 4.1. Индексирование элементов в массиве NumPy

Если при работе с многомерным массивом опустить несколько последних индексов, то будет возвращен объект ndarray меньшей размерности, содержащий данные по указанным при индексировании осям. Так, пусть имеется массив `arr3d` размерности 2 × 2 × 3:

```
In [66]: arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

In [67]: arr3d
Out[67]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

Тогда `arr3d[0]` – массив размерности 2 × 3:

```
In [68]: arr3d[0]
Out[68]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Выражению `arr3d[0]` можно присвоить как скалярное значение, так и массив:

```
In [69]: old_values = arr3d[0].copy()

In [70]: arr3d[0] = 42

In [71]: arr3d
Out[71]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])

In [72]: arr3d[0] = old_values

In [73]: arr3d
Out[73]:
```

```
array([[[ 1, 2, 3],
        [ 4, 5, 6]],
       [[ 7, 8, 9],
        [10, 11, 12]]])
```

Аналогично `arr3d[1, 0]` дает все значения, список индексов которых начинается с `(1, 0)`, т. е. одномерный массив:

```
In [74]: arr3d[1, 0]
Out[74]: array([7, 8, 9])
```

Отметим, что во всех случаях, когда выбираются участки массива, результат является представлением.

Индексирование срезами

Как и для одномерных объектов наподобие списков Python, для объектов ndarray можно формировать срезы:

```
In [75]: arr[1:6]
Out[75]: array([ 1, 2, 3, 4, 64])
```

Для объектов большей размерности вариантов больше, потому что вырезать можно по одной или нескольким осям, сочетая с выбором отдельных элементов с помощью целых индексов. Вернемся к рассмотренному выше двумерному массиву `arr2d`. Вырезание из него выглядит несколько иначе:

```
In [76]: arr2d
Out[76]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
In [77]: arr2d[:2]
Out[77]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Как видите, вырезание производится вдоль оси 0, первой оси. Поэтому срез содержит диапазон элементов вдоль этой оси. Можно указать несколько срезов – как несколько индексов:

```
In [78]: arr2d[:2, 1:]
Out[78]:
array([[2, 3],
       [5, 6]])
```

При таком вырезании мы всегда получаем представления массивов с таким же числом измерений, как у исходного. Сочетая срезы и целочисленные индексы, можно получить массивы меньшей размерности:

```
In [79]: arr2d[1, :2]
Out[79]: array([4, 5])
```

```
In [80]: arr2d[2, :1]
Out[80]: array([7])
```

Иллюстрация приведена на рис. 4.2. Отметим, что двоеточие без указания числа означает, что нужно взять всю ось целиком, поэтому для получения осей только высших размерностей можно поступить следующим образом:

```
In [81]: arr2d[:, :1]
Out[81]:
array([[1],
       [4],
       [7]])
```

Разумеется, присваивание выражению-срезу означает присваивание всем элементам этого среза:

```
In [82]: arr2d[:2, 1:] = 0
```

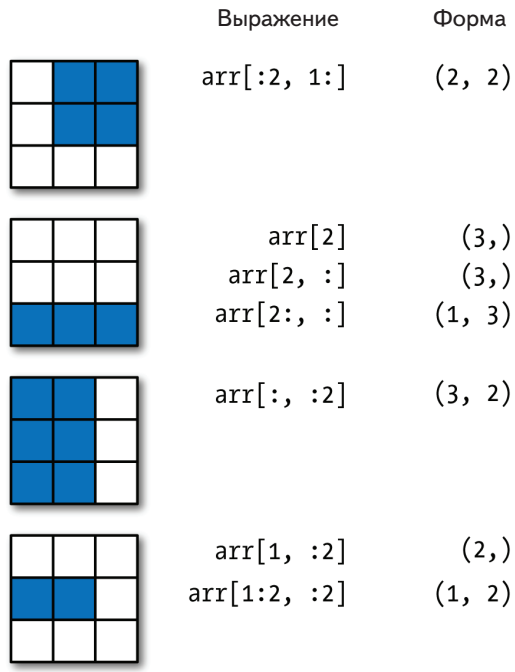


Рис. 4.2. Вырезание из двумерного массива

Булево индексирование

Пусть имеется некоторый массив с данными и массив имен, содержащий дубликаты. Я хочу воспользоваться функцией `randn` из модуля `numpy.random`, чтобы сгенерировать случайные данные с нормальным распределением:

```
In [83]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [84]: data = randn(7, 4)

In [85]: names
Out[85]:
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'],
```

```
       dtype='|S4')

In [86]: data
Out[86]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [ 2.1452,  0.8799, -0.0523,  0.0672],
       [-1.0023, -0.1698,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

Допустим, что каждое имя соответствует строке в массиве `data`, и мы хотим выбрать все строки, которым соответствует имя `'Bob'`. Операции сравнения массивов (например, `==`), как и арифметические, также векторизованы. Поэтому сравнение `names` со строкой `'Bob'` дает массив булевых величин:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

Этот булев массив можно использовать для индексирования другого массива:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 , 0.5433, -0.2349, 1.2792],
       [ 2.1452, 0.8799, -0.0523, 0.0672]])
```

Длина булева массива должна совпадать с длиной индексируемой им оси. Можно даже сочетать булевы массивы со срезами и целыми числами (или последовательностями целых чисел, о чем речь пойдет ниже):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349, 1.2792],
       [-0.0523, 0.0672]])

In [90]: data[names == 'Bob', 3]
Out[90]: array([ 1.2792, 0.0672])
```

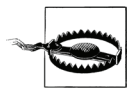
Чтобы выбрать все, кроме `'Bob'`, можно либо воспользоваться оператором сравнения `!=`, либо применить отрицание условие, обозначаемое знаком `-`:

```
In [91]: names != 'Bob'
Out[91]: array([False, True, True, False, True, True, True], dtype=bool)

In [92]: data[-(names == 'Bob')]
Out[92]:
array([[-0.268 ,  0.5465,  0.0939, -2.0445],
       [-0.047 , -2.026 ,  0.7719,  0.3103],
       [-1.0023, -0.1698,  1.1503,  1.7289],
       [ 0.1913,  0.4544,  0.4519,  0.5535],
       [ 0.5994,  0.8174, -0.9297, -1.2564]])
```

Чтобы сформировать составное булево условие, включающее два из трех имен, воспользуемся булевыми операторами `&` (И) и `|` (ИЛИ):

```
In [93]: mask = (names == 'Bob') | (names == 'Will')

In [94]: mask
Out[94]: array([True, False, True, True, True, False, False], dtype=bool)

In [95]: data[mask]
Out[95]:
array([[-0.048 ,  0.5433, -0.2349, 1.2792],
       [-0.047 , -2.026 ,  0.7719, 0.3103],
       [ 2.1452,  0.8799, -0.0523, 0.0672],
       [-1.0023, -0.1698,  1.1503, 1.7289]])
```

При выборке данных из массива путем булева индексирования *всегда* создается копия данных, даже если возвращенный массив совпадает с исходным.



Ключевые слова Python `and` и `or` с булевыми массивами не работают.

Задание значений с помощью булевых массивов работает в соответствии с ожиданиями. Чтобы заменить все отрицательные значения в массиве `data` нулем, нужно всего лишь написать:

```
In [96]: data[data < 0] = 0

In [97]: data
Out[97]:
array([[ 0.    , 0.5433, 0.    , 1.2792],
       [ 0.    , 0.5465, 0.0939, 0.    ],
       [ 0.    , 0.    , 0.7719, 0.3103],
       [ 2.1452, 0.8799, 0.    , 0.0672],
       [ 0.    , 0.    , 1.1503, 1.7289],
       [ 0.1913, 0.4544, 0.4519, 0.5535],
       [ 0.5994, 0.8174, 0.    , 0.    ]])
```

Задать целые строки или столбцы с помощью одномерного булева массива тоже просто:

```
In [98]: data[names != 'Joe'] = 7

In [99]: data
Out[99]:
array([[ 7.    , 7.    , 7.    , 7.    ],
       [ 0.    , 0.5465, 0.0939, 0.    ],
       [ 7.    , 7.    , 7.    , 7.    ],
       [ 7.    , 7.    , 7.    , 7.    ],
       [ 7.    , 7.    , 7.    , 7.    ],
       [ 0.1913, 0.4544, 0.4519, 0.5535],
       [ 0.5994, 0.8174, 0.    , 0.    ]])
```

Прихотливое индексирование

Термином *прихотливое индексирование* (fancy indexing) в NumPy обозначается индексирование с помощью целочисленных массивов. Допустим, имеется массив 8×4:

```
In [100]: arr = np.empty((8, 4))

In [101]: for i in range(8):
   .....: arr[i] = i

In [102]: arr
Out[102]:
array([[ 0., 0., 0., 0.],
       [ 1., 1., 1., 1.],
       [ 2., 2., 2., 2.],
       [ 3., 3., 3., 3.],
       [ 4., 4., 4., 4.],
       [ 5., 5., 5., 5.],
       [ 6., 6., 6., 6.],
       [ 7., 7., 7., 7.]])
```

Чтобы выбрать подмножество строк в определенном порядке, можно просто передать список или массив целых чисел, описывающих желаемый порядок:

```
In [103]: arr[[4, 3, 0, 6]]
Out[103]:
array([[ 4., 4., 4., 4.],
       [ 3., 3., 3., 3.],
       [ 0., 0., 0., 0.],
       [ 6., 6., 6., 6.]])
```

Надеюсь, что этот код делает именно то, что вы ожидаете! Если указать отрицательный индекс, то номер соответствующей строки будет отсчитываться с конца:

```
In [104]: arr[[-3, -5, -7]]
Out[104]:
array([[ 5., 5., 5., 5.],
       [ 3., 3., 3., 3.],
       [ 1., 1., 1., 1.]])
```

При передаче нескольких массивов индексов делается несколько иное: выбирается одномерный массив элементов, соответствующих каждому кортежу индексов:

```
# о функции reshape см. главу 12
In [105]: arr = np.arange(32).reshape((8, 4))

In [106]: arr
Out[106]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
```

```
        [16, 17, 18, 19],
        [20, 21, 22, 23],
        [24, 25, 26, 27],
        [28, 29, 30, 31]])

In [107]: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[107]: array([ 4, 23, 29, 10])
```

Давайте разберемся, что здесь происходит: отбираются элементы в позициях `(1, 0)`, `(5, 3)`, `(7, 1)` и `(2, 2)`. В данном случае поведение прихотливого индексирования отличается от того, что ожидают многие пользователи (я в том числе): получить прямоугольный регион, образованный подмножеством строк и столбцов матрицы. Добиться этого можно, например, так:

```
In [108]: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[108]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Другой способ – воспользоваться функцией `np.ix_`, которая преобразует два одномерных массива целых чисел в индексатор, выбирающий квадратный регион:

```
In [109]: arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
Out[109]:
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

Имейте в виду, что прихотливое индексирование, в отличие от вырезания, всегда порождает новый массив, в который копируются данные.

Транспонирование массивов и перестановка осей

Транспонирование – частный случай изменения формы, при этом также возвращается представление исходных данных без какого-либо копирования. У массивов имеется метод `transpose` и специальный атрибут `T`:

```
In [110]: arr = np.arange(15).reshape((3, 5))

In [111]: arr
Out[111]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
In [112]: arr.T
Out[112]:
array([[ 0, 5, 10],
       [ 1, 6, 11],
       [ 2, 7, 12],
       [ 3, 8, 13],
       [ 4, 9, 14]])
```

При вычислениях с матрицами эта операция применяется очень часто. Вот, например, как вычисляется матрица X^TX с помощью метода `np.dot`:

```
In [113]: arr = np.random.randn(6, 3)

In [114]: np.dot(arr.T, arr)
Out[114]:
array([[ 2.584 , 1.8753, 0.8888],
       [ 1.8753, 6.6636, 0.3884],
       [ 0.8888, 0.3884, 3.9781]])
```

Для массивов большей размерности метод `transpose` принимает кортеж номеров осей, описывающий их перестановку (чтобы ум за разум совсем заехал):

```
In [115]: arr = np.arange(16).reshape((2, 2, 4))

In [116]: arr
Out[116]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],
       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])

In [117]: arr.transpose((1, 0, 2))
Out[117]:
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],
       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Обычное транспонирование с помощью `.T` – частный случай перестановки осей. У объекта ndarray имеется метод `swapaxes`, который принимает пару номеров осей:

```
In [118]: arr
Out[118]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
In [119]: arr.swapaxes(1, 2)
Out[119]:
array([[[ 0, 4],
        [ 1, 5],
        [ 2, 6],
        [ 3, 7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

Метод `swapaxes` также возвращает представление без копирования данных.

Универсальные функции: быстрые поэлементные операции над массивами

Универсальной функцией, или *u-функцией* называется функция, которая выполняет поэлементные операции над данными, хранящимися в объектах ndarray. Можно считать, что это векторные обертки вокруг простых функций, которые

принимают одно или несколько скалярных значений и порождают один или несколько скалярных результатов.

Многие u-функции – простые поэлементные преобразования, например `sqrt` или `exp`:

```
In [120]: arr = np.arange(10)

In [121]: np.sqrt(arr)
Out[121]:
array([ 0.    , 1.    , 1.4142, 1.7321, 2. , 2.2361, 2.4495,
        2.6458, 2.8284, 3.    ])

In [122]: np.exp(arr)
Out[122]:
array([ 1.      ,   2.7183,    7.3891,   20.0855,   54.5982,
        148.4132, 403.4288, 1096.6332, 2980.958 , 8103.0839])
```

Такие u-функции называются *унарными*. Другие, например `add` или `maximum`, принимают 2 массива (и потому называются *бинарными*) и возвращают один результирующий массив:

```
In [123]: x = randn(8)
In [124]: y = randn(8)
In [125]: x
Out[125]:
array([ 0.0749,  0.0974,  0.2002, -0.2551,  0.4655, 0.9222, 0.446 ,
       -0.9337])

In [126]: y
Out[126]:
array([ 0.267 , -1.1131, -0.3361,  0.6117, -1.2323, 0.4788, 0.4315,
       -0.7147])

In [127]: np.maximum(x, y) # поэлементный максимум
Out[127]:
array([ 0.267 ,  0.0974,  0.2002,  0.6117,  0.4655, 0.9222, 0.446 ,
       -0.7147])
```

Хотя и нечасто, но можно встретить u-функцию, возвращающую несколько массивов. Примером может служить `modf`, векторный вариант встроенной в Python функции `divmod`: она возвращает дробные и целые части хранящихся в массиве чисел с плавающей точкой:

```
In [128]: arr = randn(7) * 5

In [129]: np.modf(arr)
Out[129]:
(array([-0.6808, 0.0636, -0.386 , 0.1393, -0.8806, 0.9363, -0.883 ]),
 array([-2., 4., -3., 5., -3., 3., -6.]))
```

В таблицах 4.3 и 4.4 перечислены имеющиеся u-функции.

Таблица 4.3. Унарные u-функции

Функция	Описание
`abs, fabs`	Вычислить абсолютное значение целых, вещественных или комплексных элементов массива. Для вещественных данных `fabs` работает быстрее
`sqrt`	Вычислить квадратный корень из каждого элемента. Эквивалентно `arr ** 0.5`
`square`	Вычислить квадрат каждого элемента. Эквивалентно `arr ** 2`
`exp`	Вычислить экспоненту e^x каждого элемента
`log, log10, log2, log1p`	Натуральный (по основанию `e`), десятичный, двоичный логарифм и функция `log(1 + x)` соответственно
`sign`	Вычислить знак каждого элемента: 1 (для положительных чисел), 0 (для нуля) или –1 (для отрицательных чисел)
`ceil`	Вычислить для каждого элемента наименьшее целое число, не меньшее его
`floor`	Вычислить для каждого элемента наибольшее целое число, не большее его
`rint`	Округлить элементы до ближайшего целого с сохранением dtype
`modf`	Вернуть дробные и целые части массива в виде отдельных массивов
`isnan`	Вернуть булев массив, показывающий, какие значения являются NaN (не числами)
`isfinite, isinf`	Вернуть булев массив, показывающий, какие элементы являются конечными (не inf и не NaN) или бесконечными соответственно
`cos, cosh, sin, sinh, tan, tanh`	Обычные и гиперболические тригонометрические функции
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Обратные тригонометрические функции
`logical_not`	Вычислить значение истинности `not x` для каждого элемента. Эквивалентно `-arr`

Таблица 4.4. Бинарные u-функции

Функция	Описание
`add`	Сложить соответственные элементы массивов
`subtract`	Вычесть элементы второго массива из соответственных элементов первого
`multiply`	Перемножить соответственные элементы массивов
`divide, floor_divide`	Деление и деление с отбрасыванием остатка

Функция	Описание
power	Возвести элементы первого массива в степени, указанные во втором массиве
maximum, fmax	Поэлементный максимум. Функция fmax игнорирует значения NaN
minimum, fmin	Поэлементный минимум. Функция fmin игнорирует значения NaN
mod	Поэлементный модуль (остаток от деления)
copysign	Копировать знаки значений второго массива в соответственные элементы первого массива
greater, greater_equal, less, less_equal, equal, not_equal	Поэлементное сравнение, возвращается булев массив. Эквивалентны инфиксным операторам >, >=, <, <=, ==, !=
logical_and, logical_or, logical_xor	Вычислить логическое значение истинности логических операций. Эквивалентны инфиксным операторам &, |, ^

Обработка данных с применением массивов

С помощью массивов NumPy многие виды обработки данных можно записать очень кратко, не прибегая к циклам. Такой способ замены явных циклов выражениями-массивами обычно называется *векторизацией*. Вообще говоря, векторные операции с массивами выполняются на один-два (а то и больше) порядка быстрее, чем эквивалентные операции на чистом Python. Позже, в главе 12 я расскажу об *укладывании*, действенном методе векторизации вычислений.

В качестве простого примера предположим, что нужно вычислить функцию sqrt(x^2 + y^2) на регулярной сетке. Функция np.meshgrid принимает два одномерных массива и порождает две двумерные матрицы, соответствующие всем парам (x, y) элементов, взятых из обоих массивов:

```
In [130]: points = np.arange(-5, 5, 0.01) # 1000 равноотстоящих точек

In [131]: xs, ys = np.meshgrid(points, points)

In [132]: ys
Out[132]:
array([[-5.  , -5.  , -5.  , ..., -5.  , -5.  , -5.  ],
       [-4.99, -4.99, -4.99, ..., -4.99, -4.99, -4.99],
       [-4.98, -4.98, -4.98, ..., -4.98, -4.98, -4.98],
       ...,
       [ 4.97,  4.97,  4.97, ...,  4.97,  4.97,  4.97],
       [ 4.98,  4.98,  4.98, ...,  4.98,  4.98,  4.98],
       [ 4.99,  4.99,  4.99, ...,  4.99,  4.99,  4.99]])
```

Теперь для вычисления функции достаточно написать такое же выражение, как для двух точек:

```
In [134]: import matplotlib.pyplot as plt

In [135]: z = np.sqrt(xs ** 2 + ys ** 2)

In [136]: z

Out[136]:
array([[ 7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
       [ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
       [ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       ...,
       [ 7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
       [ 7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
       [ 7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])

In [137]: plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
Out[137]: <matplotlib.colorbar.Colorbar instance at 0x4e46d40>

In [138]: plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
Out[138]: <matplotlib.text.Text at 0x4565790>
```

На рис. 4.3 показан результат применения функции `imshow` из библиотеки matplotlib для создания изображения по двумерному массиву значений функции.

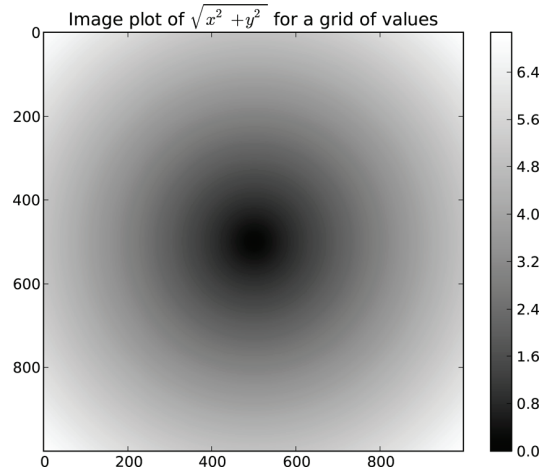


Рис. 4.3. График функции двух переменных на сетке

Запись логических условий в виде операций с массивами

Функция `numpy.where` – это векторный вариант тернарного выражения `x if condition else y`. Пусть есть булев массив и два массива значений:

```
In [140]: xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

In [141]: yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

In [142]: cond = np.array([True, False, True, True, False])
```

Допустим, что мы хотим брать значение из массива `xarr`, если соответственное значение в массиве `cond` равно `True`, а в противном случае – значение из `yarr`. Эту задачу решает такая операция спискового включения:

```
In [143]: result = [(x if c else y)
   .....: for x, y, c in zip(xarr, yarr, cond)]

In [144]: result
Out[144]: [1.1000000000000001, 2.2000000000000002, 1.3, 1.3999999999999999, 2.5]
```

Здесь сразу несколько проблем. Во-первых, для больших массивов это будет не быстро (потому что весь код написан на чистом Python). Во-вторых, к многомерным массивам такое решение вообще неприменимо. С помощью функции `np.where` можно написать очень лаконичный код:

```
In [145]: result = np.where(cond, xarr, yarr)

In [146]: result
Out[146]: array([ 1.1, 2.2, 1.3, 1.4, 2.5])
```

Второй и третий аргументы `np.where` не обязаны быть массивами – один или оба могут быть скалярами. При анализе данные `where` обычно применяется, чтобы создать новый массив на основе существующего. Предположим, имеется матрица со случайными данными, и мы хотим заменить все положительные значение на 2, а все отрицательные – на –2. С помощью `np.where` сделать это очень просто:

```
In [147]: arr = randn(4, 4)
In [148]: arr

Out[148]:
array([[ 0.6372,  2.2043, 1.7904,  0.0752],
       [-1.5926, -1.1536, 0.4413,  0.3483],
       [-0.1798,  0.3299, 0.7827, -0.7585],
       [ 0.5857,  0.1619, 1.3583, -1.3865]])

In [149]: np.where(arr > 0, 2, -2)
Out[149]:
array([[ 2,  2, 2,  2],
       [-2, -2, 2,  2],
       [-2,  2, 2, -2],
       [ 2,  2, 2, -2]])

In [150]: np.where(arr > 0, 2, arr) # только положительные заменить на 2
Out[150]:
array([[    2. ,     2. , 2. ,     2. ],
       [-1.5926, -1.1536, 2. ,     2. ],
```

```
       [-0.1798,     2. , 2. , -0.7585],
       [    2. ,     2. , 2. , -1.3865]])
```

Передавать `where` можно не только массивы одинакового размера или скаляры. При некоторой изобретательности `where` позволяет выразить и более сложную логику. Пусть есть два булева массива `cond1` и `cond2`, и мы хотим сопоставить разные значения каждой из четырех возможных комбинаций двух булевых значений:

```
result = []
for i in range(n):
  if cond1[i] and cond2[i]:
     result.append(0)
  elif cond1[i]:
     result.append(1)
  elif cond2[i]:
     result.append(2)
  else:
     result.append(3)
```

Хотя сразу это не очевидно, показанный цикл `for` можно преобразовать во вложенное выражение `where`:

```
np.where(cond1 & cond2, 0,
         np.where(cond1, 1,
                  np.where(cond2, 2, 3)))
```

В этом конкретном примере можно также воспользоваться тем фактом, что во всех вычислениях булевы значения трактуются как 0 и 1, поэтому выражение можно записать и в виде такой арифметической операции (хотя выглядит она загадочно):

```
result = 1 * cond1 + 2 * cond2 + 3 * -(cond1 | cond2)
```

Математические и статистические операции

Среди методов массива есть математические функции, которые вычисляют статистики массива в целом или данных вдоль одной оси. Выполнить агрегирование (часто его называют *редукцией*) типа `sum`, `mean` или стандартного отклонения `std` можно как с помощью метода экземпляра массива, так и функции на верхнем уровне NumPy:

```
In [151]: arr = np.random.randn(5, 4) # нормально распределенные данные

In [152]: arr.mean()
Out[152]: 0.062814911084854597

In [153]: np.mean(arr)
Out[153]: 0.062814911084854597

In [154]: arr.sum()
Out[154]: 1.2562982216970919
```

Функции типа `mean` и `sum` принимают необязательный аргумент `axis`, при наличии которого вычисляется статистика по заданной оси, и в результате порождается массив на единицу меньшей размерности:

```
In [155]: arr.mean(axis=1)
Out[155]: array([-1.2833, 0.2844, 0.6574, 0.6743, -0.0187])

In [156]: arr.sum(0)
Out[156]: array([-3.1003, -1.6189, 1.4044, 4.5712])
```

Другие методы, например `cumsum` и `cumprod`, ничего не агрегируют, а порождают массив промежуточных результатов:

```
In [157]: arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])

In [158]: arr.cumsum(0)
Out[158]:
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])

In [159]: arr.cumprod(1)
Out[159]:
array([[ 0,  0,   0],
       [ 3, 12,  60],
       [ 6, 42, 336]])
```

Полный список приведен в табл. 4.5. Как многие из этих методов применяются на практике, мы увидим в последующих главах.

Таблица 4.5. Статистические методы массива

Метод	Описание
`sum`	Сумма элементов всего массива или вдоль одной оси. Для массивов нулевой длины функция `sum` возвращает 0
`mean`	Среднее арифметическое. Для массивов нулевой длины равно NaN
`std, var`	Стандартное отклонение и дисперсия, соответственно. Может быть задано число степеней свободы (по умолчанию знаменатель равен `n`)
`min, max`	Минимум и максимум
`argmin, argmax`	Индексы минимального и максимального элемента
`cumsum`	Нарастающая сумма с начальным значением 0
`cumprod`	Нарастающее произведение с начальным значением 1

Методы булевых массивов

В вышеупомянутых методах булевы значения приводятся к 1 (`True`) и 0 (`False`). Поэтому функция `sum` часто используется для подсчета значений `True` в булевом массиве:

```
In [160]: arr = randn(100)

In [161]: (arr > 0).sum() # Количество положительных значений
Out[161]: 44
```

Но существуют еще два метода, `any` и `all`, особенно полезных в случае булевых массивов. Метод `any` проверяет, есть ли в массиве хотя бы одно значение, равное `True`, а `all` – что все значения в массиве равны `True`:

```
In [162]: bools = np.array([False, False, True, False])

In [163]: bools.any()
Out[163]: True

In [164]: bools.all()
Out[164]: False
```

Эти методы работают и для небулевых массивов, и тогда все отличные от нуля элементы считаются равными `True`.

Сортировка

Как и встроенные в Python списки, массивы NumPy можно сортировать на месте методом `sort`:

```
In [165]: arr = randn(8)

In [166]: arr
Out[166]:
array([ 0.6903, 0.4678, 0.0968, -0.1349, 0.9879, 0.0185, -1.3147,
       -0.5425])

In [167]: arr.sort()

In [168]: arr
Out[168]:

array([-1.3147, -0.5425, -0.1349, 0.0185, 0.0968, 0.4678, 0.6903,
        0.9879])
```

В многомерных массивах можно сортировать на месте одномерные секции вдоль любой оси, для этого нужно передать `sort` номер оси:

```
In [169]: arr = randn(5, 3)

In [170]: arr
Out[170]:
array([[-0.7139, -1.6331, -0.4959],
       [ 0.8236, -1.3132, -0.1935],
       [-1.6748,  3.0336, -0.863 ],
       [-0.3161,  0.5362, -2.468 ],
       [ 0.9058,  1.1184, -1.0516]])

In [171]: arr.sort(1)

In [172]: arr
Out[172]:
array([[-1.6331, -0.7139, -0.4959],
```

```
       [-1.3132, -0.1935,  0.8236],
       [-1.6748, -0.863 ,  3.0336],
       [-2.468 , -0.3161,  0.5362],
       [-1.0516,  0.9058,  1.1184]])
```

Метод верхнего уровня `np.sort` возвращает отсортированную копию массива, а не сортирует массив на месте. Чтобы, не мудрствуя лукаво, вычислить квантили массива, нужно отсортировать его и выбрать значение с конкретным рангом:

```
In [173]: large_arr = randn(1000)
In [174]: large_arr.sort()

In [175]: large_arr[int(0.05 * len(large_arr))] # 5%-ный квантиль
Out[175]: -1.5791023260896004
```

Дополнительные сведения о методах сортировки в NumPy и о более сложных приемах, например косвенной сортировке, см. главу 12. В библиотеке pandas есть еще несколько операций, относящихся к сортировке (например, сортировка таблицы по одному или нескольким столбцам).

Устранение дубликатов и другие теоретико-множественные операции

В NumPy имеется основные теоретико-множественные операции для одномерных массивов. Пожалуй, самой употребительной является `np.unique`, она возвращает отсортированное множество уникальных значений в массиве:

```
In [176]: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

In [177]: np.unique(names)
Out[177]:
array(['Bob', 'Joe', 'Will'],
dtype='|S4')

In [178]: ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])

In [179]: np.unique(ints)
Out[179]: array([1, 2, 3, 4])
```

Сравните `np.unique` с альтернативой на чистом Python:

```
In [180]: sorted(set(names))
Out[180]: ['Bob', 'Joe', 'Will']
```

Функция `np.in1d` проверяет, присутствуют ли значения из одного массива в другом, и возвращает булев массив:

```
In [181]: values = np.array([6, 0, 0, 3, 2, 5, 6])

In [182]: np.in1d(values, [2, 3, 6])
Out[182]: array([ True, False, False, True, True, False, True], dtype=bool)
```

В табл. 4.6 перечислены все теоретико-множественные функции, имеющиеся в NumPy.

Таблица 4.6. Теоретико-множественные операции с массивами

Метод	Описание
`unique(x)`	Вычисляет отсортированное множество уникальных элементов
`intersect1d(x, y)`	Вычисляет отсортированное множество элементов, общих для `x` и `y`
`union1d(x, y)`	Вычисляет отсортированное объединение элементов
`in1d(x, y)`	Вычисляет булев массив, показывающий, какие элементы `x` встречаются в `y`
`setdiff1d(x, y)`	Вычисляет разность множеств, т. е. элементы, принадлежащие `x`, но не принадлежащие `y`
`setxor1d(x, y)`	Симметрическая разность множеств; элементы, принадлежащие одному массиву, но не обоим сразу

Файловый ввод-вывод массивов

NumPy умеет сохранять на диске и загружать с диска данные в текстовом или двоичном формате. В последующих главах мы узнаем, как в pandas считываются в память табличные данные.

Хранение массивов на диске в двоичном формате

`np.save` и `np.load` – основные функции для эффективного сохранения и загрузки данных с диска. По умолчанию массивы хранятся в несжатом двоичном формате в файле с расширением `.npy`.

```
In [183]: arr = np.arange(10)

In [184]: np.save('some_array', arr)
```

Если путь к файлу не заканчивается суффиксом `.npy`, то он будет добавлен. Хранящийся на диске массив можно загрузить в память функцией `np.load`:

```
In [185]: np.load('some_array.npy')
Out[185]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Можно сохранить несколько массивов в zip-архиве с помощью функции `np.savez`, которой массивы передаются в виде именованных аргументов:

```
In [186]: np.savez('array_archive.npz', a=arr, b=arr)
```

При считывании `npz`-файла мы получаем похожий на словарь объект, который отложенно загружает отдельные массивы:

```
In [187]: arch = np.load('array_archive.npz')

In [188]: arch['b']
Out[188]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Сохранение и загрузка текстовых файлов

Загрузка текста из файлов – вполне стандартная задача. Многообразие имеющихся в Python функций для чтения и записи файлов может смутить начинающего, поэтому я буду рассматривать главным образом функции `read_csv` и `read_table` из библиотеки pandas. Иногда бывает полезно загружать данные в массивы NumPy с помощью функции `np.loadtxt` или более специализированной `np.genfromtxt`.

У этих функций много параметров, которые позволяют задавать разные разделители и функции-конвертеры для столбцов, пропускать строки и делать много других вещей. Рассмотрим простой случай загрузки файла с разделителями-запятыми (CSV):

```
In [191]: !cat array_ex.txt
0.580052,0.186730,1.040717,1.134411
0.194163,-0.636917,-0.938659,0.124094
-0.126410,0.268607,-0.695724,0.047428
-1.484413,0.004176,-0.744203,0.005487
2.302869,0.200131,1.670238,-1.881090
-0.193230,1.047233,0.482803,0.960334
```

Его можно следующим образом загрузить в двумерный массив:

```
In [192]: arr = np.loadtxt('array_ex.txt', delimiter=',')

In [193]: arr
Out[193]:
array([[ 0.5801,  0.1867,  1.0407,  1.1344],
       [ 0.1942, -0.6369, -0.9387,  0.1241],
       [-0.1264,  0.2686, -0.6957,  0.0474],
       [-1.4844,  0.0042, -0.7442,  0.0055],
       [ 2.3029,  0.2001,  1.6702, -1.8811],
       [-0.1932,  1.0472,  0.4828,  0.9603]])
```

Функция `np.savetxt` выполняет обратную операцию: записывает массив в текстовый файл с разделителями. Функция `genfromtxt` аналогична `loadtxt`, но ориентирована на структурные массивы и обработку отсутствующих данных. Подробнее о структурных массивах см. главу 12.

Дополнительные сведения о чтении и записи файлов, особенно табличных, приведены в последующих главах, касающихся библиотеки pandas и объектов DataFrame.

Линейная алгебра

Операции линейной алгебры – умножение и разложение матриц, вычисление определителей и другие – важная часть любой библиотеки для работы с массивами. В отличие от некоторых языков, например MATLAB, в NumPy применение оператора * к двум двумерным массивам вычисляет поэлементное, а не матричное произведение. А для перемножения матриц имеется функция `dot` – как в виде метода массива, так и в виде функции в пространстве имен `numpy`:

```
In [194]: x = np.array([[1., 2., 3.], [4., 5., 6.]])

In [195]: y = np.array([[6., 23.], [-1, 7], [8, 9]])

In [196]: x                              In [197]: y
Out[196]:                                Out[197]:
array([[ 1., 2., 3.],                   array([[  6., 23.],
       [ 4., 5., 6.]])                          [ -1.,  7.],
                                                [  8.,  9.]])
In [198]: x.dot(y)      # эквивалентно np.dot(x, y)
Out[198]:
array([[ 28.,  64.],
       [ 67., 181.]])
```

Произведение двумерного массива и одномерного массива подходящего размера дает одномерный массив:

```
In [199]: np.dot(x, np.ones(3))
Out[199]: array([ 6., 15.])
```

В модуле `numpy.linalg` имеет стандартный набор алгоритмов, в частности, разложение матриц, нахождение обратной матрицы и вычисление определителя. Все они реализованы на базе тех же отраслевых библиотек, написанных на Fortran, которые используются и в других языках, например MATLAB и R: BLAS, LAPACK и, возможно (в зависимости от сборки NumPy), библиотеки MKL, поставляемой компанией Intel:

```
In [201]: from numpy.linalg import inv, qr

In [202]: X = randn(5, 5)

In [203]: mat = X.T.dot(X)

In [204]: inv(mat)
Out[204]:
array([[ 3.0361, -0.1808, -0.6878, -2.8285, -1.1911],
       [-0.1808,  0.5035,  0.1215,  0.6702,  0.0956],
       [-0.6878,  0.1215,  0.2904,  0.8081,  0.3049],
       [-2.8285,  0.6702,  0.8081,  3.4152,  1.1557],
       [-1.1911,  0.0956,  0.3049,  1.1557,  0.6051]])

In [205]: mat.dot(inv(mat))
Out[205]:
```

```
array([[ 1.,  0.,  0., 0., -0.],
       [ 0.,  1., -0., 0.,  0.],
       [ 0., -0.,  1., 0.,  0.],
       [ 0., -0., -0., 1., -0.],
       [ 0.,  0.,  0., 0.,  1.]])

In [206]: q, r = qr(mat)

In [207]: r
Out[207]:
array([[ -6.9271,  7.389 ,   6.1227, -7.1163, -4.9215],
       [ 0.    , -3.9735,  -0.8671,  2.9747, -5.7402],
       [ 0.    ,  0.    , -10.2681,  1.8909,  1.6079],
       [ 0.    ,  0.    ,   0.    , -1.2996,  3.3577],
       [ 0.    ,  0.    ,   0.    ,  0.    ,  0.5571]])
```

В табл. 4.7 перечислены наиболее употребительные функции линейной алгебры.

Таблица 4.7. Наиболее употребительные функции из модуля `numpy.linalg`

Функция	Описание
`diag`	Возвращает диагональные элементы квадратной матрицы в виде одномерного массива или преобразует одномерный массив в квадратную матрицу, в которой все элементы, кроме находящихся на главной диагонали, равны нулю
`dot`	Вычисляет произведение матриц
`trace`	Вычисляет след матрицы – сумму диагональных элементов
`det`	Вычисляет определитель матрицы
`eig`	Вычисляет собственные значения и собственные векторы квадратной матрицы
`inv`	Вычисляет обратную матрицу
`pinv`	Вычисляет псевдообратную матрицу Мура-Пенроуза для квадратной матрицы
`qr`	Вычисляет QR-разложение
`svd`	Вычисляет сингулярное разложение (SVD)
`solve`	Решает линейную систему `Ax = b`, где `A` – квадратная матрица
`lstsq`	Вычисляет решение уравнения `y = Xb` по методу наименьших квадратов

Генерация случайных чисел

Модуль `numpy.random` дополняет встроенный модуль `random` функциями, которые генерируют целые массивы случайных чисел с различными распределениями вероятности. Например, с помощью функции можно получить случайный массив 4×4 с нормальным распределением:

```
In [208]: samples = np.random.normal(size=(4, 4))

In [209]: samples
Out[209]:
```

```
array([[ 0.1241,  0.3026,  0.5238,  0.0009],
       [ 1.3438, -0.7135, -0.8312, -2.3702],
       [-1.8608, -0.8608,  0.5601, -1.2659],
       [ 0.1198, -1.0635,  0.3329, -2.3594]])
```

Встроенный в Python модуль `random` умеет выдавать только по одному случайному числу за одно обращение. Ниже видно, что `numpy.random` более чем на порядок быстрее стандартного модуля при генерации очень больших выборок:

```
In [210]: from random import normalvariate

In [211]: N = 1000000

In [212]: %timeit samples = [normalvariate(0, 1) for _ in xrange(N)]
1 loops, best of 3: 1.33 s per loop

In [213]: %timeit np.random.normal(size=N)
10 loops, best of 3: 57.7 ms per loop
```

В табл. 4.8 приведен неполный перечень функций, имеющихся в модуле `numpy.random`. В следующем разделе я приведу несколько примеров их использования для генерации больших случайных массивов.

Таблица 4.8. Наиболее употребительные функции из модуля `numpy.random`

Функция	Описание
`seed`	Задает начальное значение генератора случайных чисел
`permutation`	Возвращает случайную перестановку последовательности или диапазона
`shuffle`	Случайным образом переставляет последовательность на месте
`rand`	Случайная выборка с равномерным распределением
`randint`	Случайная выборка целого числа из заданного диапазона
`randn`	Случайная выборка с нормальным распределением со средним 0 и стандартным отклонением 1 (интерфейс похож на MATLAB)
`binomial`	Случайная выборка с биномиальным распределением
`normal`	Случайная выборка с нормальным (гауссовым) распределением
`beta`	Случайная выборка с бета-распределением
`chisquare`	Случайная выборка с распределением хи-квадрат
`gamma`	Случайная выборка с гамма-распределением
`uniform`	Случайная выборка с равномерным распределением на полуинтервале [0, 1)

Пример: случайное блуждание

Проиллюстрируем операции с массивами на примере случайного блуждания. Сначала рассмотрим случайное блуждание с начальной точкой 0 и шагами 1 и –1, выбираемыми с одинаковой вероятностью. Вот реализация одного случайного блуждания с 1000 шагами на чистом Python с помощью встроенного модуля `random`:

```
import random
position = 0
walk = [position]
steps = 1000
for i in xrange(steps):
  step = 1 if random.randint(0, 1) else -1
  position += step
  walk.append(position)
```

На рис. 4.4 показаны первые 100 значений такого случайного блуждания.

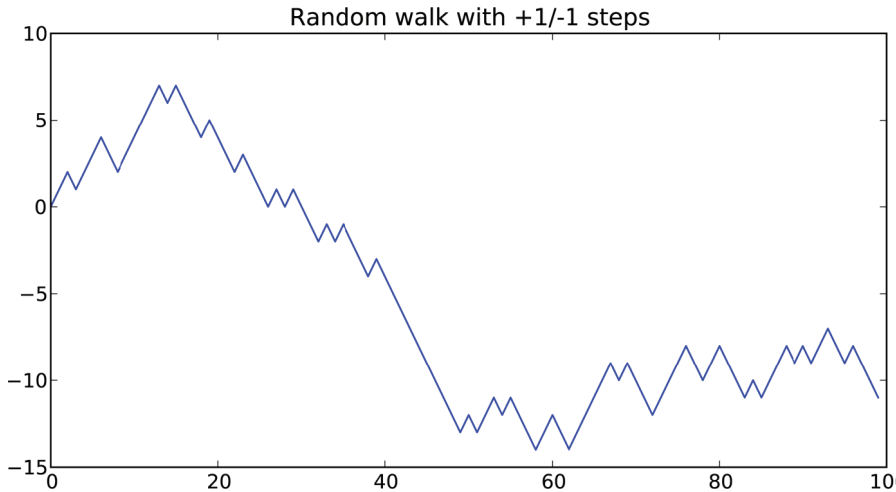


Рис. 4.4. Простое случайное блуждание

Наверное, вы обратили внимание, что `walk` – это просто нарастающая сумма случайных шагов, которую можно вычислить как выражение-массив. Поэтому я воспользуюсь модулем `np.random`, чтобы за один присест подбросить 1000 монет с исходами 1 и –1 и вычислить нарастающую сумму:

```
In [215]: nsteps = 1000

In [216]: draws = np.random.randint(0, 2, size=nsteps)

In [217]: steps = np.where(draws > 0, 1, -1)

In [218]: walk = steps.cumsum()
```

Теперь можно приступить к вычислению статистики, например минимального и максимального значения на траектории блуждания:

```
In [219]: walk.min()          In [220]: walk.max()
Out[219]: -3                  Out[220]: 31
```

Более сложная статистика – *момент первого пересечения* – это шаг, на котором траектория случайного блуждания впервые достигает заданного значения.

В данном случае мы хотим знать, сколько времени потребуется на то, чтобы удалиться от начала (нуля) на десять единиц в любом направлении. Выражение `np.abs(walk) >= 10` дает булев массив, показывающий, в какие моменты блуждание достигало или превышало 10, однако нас интересует индекс *первого* значения 10 или –10. Его можно вычислить с помощью функции `argmax`, которая возвращает индекс первого максимального значения в булевом массиве (`True` – максимальное значение):

```
In [221]: (np.abs(walk) >= 10).argmax()
Out[221]: 37
```

Отметим, что использование здесь `argmax` не всегда эффективно, потому что она всегда просматривает весь массив. В данном частном случае мы знаем, что первое же встретившееся значение `True` является максимальным.

Моделирование сразу нескольких случайных блужданий

Если бы нам требовалось смоделировать много случайных блужданий, скажем 5000, то это можно было бы сделать путем совсем небольшой модификации приведенного выше кода. Если функциям из модуля `numpy.random` передать 2-кортеж, то они сгенерируют двумерный массив случайных чисел, и мы сможем вычислить нарастающие суммы по строкам, т. е. все 5000 случайных блужданий за одну операцию:

```
In [222]: nwalks = 5000

In [223]: nsteps = 1000

In [224]: draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # 0 or 1

In [225]: steps = np.where(draws > 0, 1, -1)

In [226]: walks = steps.cumsum(1)

In [227]: walks
Out[227]:
array([[  1,  0,  1, ...,   8,   7,   8],
       [  1,  0, -1, ...,  34,  33,  32],
       [  1,  0, -1, ...,   4,   5,   4],
       ...,
       [  1,  2,  1, ...,  24,  25,  26],
       [  1,  2,  3, ...,  14,  13,  14],
       [ -1, -2, -3, ..., -24, -23, -22]])
```

Теперь мы можем вычислить максимум и минимум по всем блужданиям:

```
In [228]: walks.max()          In [229]: walks.min()
Out[228]: 138                  Out[229]: -133
```

Вычислим для этих блужданий минимальный момент первого пересечения с уровнем 30 или -30. Это не так просто, потому что не в каждом блуждании уровень 30 достигается. Проверить, так ли это, можно с помощью метода `any`:

```
In [230]: hits30 = (np.abs(walks) >= 30).any(1)

In [231]: hits30
Out[231]: array([False, True, False, ..., False, True, False], dtype=bool)

In [232]: hits30.sum()         # Сколько раз достигалось 30 или -30
Out[232]: 3410
```

Имея этот булев массив, мы можем выбрать те строки `walks`, в которых достигается уровень 30 (по абсолютной величине), и вызвать `argmax` вдоль оси 1 для получения моментов пересечения:

```
In [233]: crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)

In [234]: crossing_times.mean()
Out[234]: 498.88973607038122
```

Поэкспериментируйте с другими распределениями шагов, не ограничиваясь подбрасыванием правильной монеты. Всего-то и нужно, что взять другую функцию генерации случайных чисел, например, `normal` для генерации шагов с нормальным распределением с заданными средним и стандартным отклонением:

```
In [235]: steps = np.random.normal(loc=0, scale=0.25,
   .....:                          size=(nwalks, nsteps))
```

ГЛАВА 5.
Первое знакомство с pandas

Библиотека pandas будет основным предметом изучения в последующих главах. Она содержит высокоуровневые структуры данных и средства манипуляции ими, спроектированные так, чтобы обеспечить простоту и высокую скорость анализа данных на Python. Эта библиотека построена поверх NumPy, поэтому ей легко пользоваться в приложениях, ориентированных на NumPy.

Для сведения – я начал разрабатывать pandas в начале 2008 года, когда работал в компании AQR, занимающейся количественным анализом инвестиционных рисков. Тогда я столкнулся с требованиями, которые не мог полностью удовлетворить ни один из имевшихся в моем распоряжении инструментов:

- Структуры данных с помеченными осями, поддерживающие автоматическое или явное выравнивание. Это помогает избежать типичных ошибок, связанных с невыровненностью данных и работой с данными, поступившими из разных источников и по-разному проиндексированных.
- Встроенная функциональность для работы с временными рядами.
- Одни и те же структуры должны быть пригодны для обработки как временных рядов, так и данных иного характера.
- Арифметические операции и операции редуцирования (например, суммирование вдоль оси) должны «пробрасывать» метаданные (метки осей).
- Гибкая обработка отсутствующих данных.
- Объединение и другие реляционные операции, имеющиеся в популярных базах данных (например, на основе SQL).

Я хотел, чтобы все это было сосредоточено в одном месте и предпочтительно написано на языке, подходящем для разработки программ общего назначения. Python казался неплохим кандидатом, но в то время в нем не было встроенных структур данных и инструментов, поддерживающих нужную мне функциональность.

За прошедшие четыре года pandas превратилась в довольно обширную библиотеку, способную решать куда более широкий круг задач обработки данных, чем я планировал первоначально, но расширялась она, не принося в жертву простоту и удобство использования, к которым я стремился с самого начала. Надеюсь, что, прочитав эту книгу, вы так же, как и я, станете считать ее незаменимым инструментом.

В этой книге используются следующие соглашения об импорте для pandas:

```
In [1]: from pandas import Series, DataFrame

In [2]: import pandas as pd
```

Таким образом, увидев в коде строку `pd.`, знайте, что это ссылка на pandas. Объекты Series и DataFrame используются так часто, что я счел полезным импортировать их в локальное пространство имен.

Введение в структуры данных pandas

Чтобы начать работу с pandas, вы должны освоить две основные структуры данных: *Series* и *DataFrame*. Они, конечно, не являются универсальным решением любой задачи, но все же образуют солидную и простую для использования основу большинства приложений.

Объект Series

Series – одномерный похожий на массив объект, содержащий массив данных (любого типа, поддерживаемого NumPy) и ассоциированный с ним массив меток, который называется *индексом*. Простейший объект Series состоит только из массива данных:

```
In [4]: obj = Series([4, 7, -5, 3])

In [5]: obj
Out[5]:
0    4
1    7
2   -5
3    3
```

В строковом представлении Series, отображаемом в интерактивном режиме, индекс находится слева, а значения справа. Поскольку мы не задали индекс для данных, то по умолчанию создается индекс, состоящий из целых чисел от 0 до $N - 1$ (где N – длина массива данных). Имея объект Series, получить представление самого массива и его индекса можно с помощью атрибутов `values` и `index` соответственно:

```
In [6]: obj.values
Out[6]: array([ 4, 7, -5, 3])

In [7]: obj.index
Out[7]: Int64Index([0, 1, 2, 3])
```

Часто желательно создать объект Series с индексом, идентифицирующим каждый элемент данных:

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [9]: obj2
Out[9]:
d    4
b    7
a   -5
c    3

In [10]: obj2.index
Out[10]: Index([d, b, a, c], dtype=object)
```

В отличие от обычного массива NumPy, для выборки одного или нескольких элементов из объекта Series можно использовать значения индекса:

```
In [11]: obj2['a']
Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]
Out[13]:
c    3
a   -5
d    6
```

Операции с массивом NumPy, например фильтрация с помощью булева массива, скалярное умножение или применение математических функций, сохраняют связь между индексом и значением:

```
In [14]: obj2
Out[14]:
d    6
b    7
a   -5
c    3

In [15]: obj2[obj2 > 0]
Out[15]:
d    6
b    7
c    3

In [16]: obj2 * 2
Out[16]:
d    12
b    14
a   -10
c     6

In [17]: np.exp(obj2)
Out[17]:
d    403.428793
b   1096.633158
a      0.006738
c     20.085537
```

Объект Series можно также представлять себе как упорядоченный словарь фиксированной длины, поскольку он отображает индекс на данные. Его можно передавать многим функциям, ожидающим получить словарь:

```
In [18]: 'b' in obj2
Out[18]: True

In [19]: 'e' in obj2
Out[19]: False
```

Если имеется словарь Python, содержащий данные, то из него можно создать объект Series:

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [21]: obj3 = Series(sdata)

In [22]: obj3
Out[22]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
```

Если передается только словарь, то в получившемся объекте Series ключи будут храниться в индексе по порядку:

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [24]: obj4 = Series(sdata, index=states)

In [25]: obj4
Out[25]:
California    NaN
Ohio        35000
Oregon      16000
Texas       71000
```

В данном случае 3 значения, найденные в `sdata`, помещены в соответствующие им позиции, а для метки `'California'` никакого значения не нашлось, поэтому ей соответствует признак `NaN` (не число), которым в pandas обозначаются отсутствующие значения. Иногда, говоря об отсутствующих данных, я буду употреблять термин «NA». Для распознавания отсутствующих данных в pandas следует использовать функции `isnull` и `notnull`:

```
In [26]: pd.isnull(obj4)
Out[26]:
California     True
Ohio          False
Oregon        False
Texas         False
```

```
In [27]: pd.notnull(obj4)
Out[27]:
California    False
Ohio           True
Oregon         True
Texas          True
```

У объекта Series есть также методы экземпляра:

```
In [28]: obj4.isnull()
Out[28]:
California     True
Ohio          False
Oregon        False
Texas         False
```

Более подробно работа с отсутствующими данными будет обсуждаться ниже в этой главе. Для многих приложений особенно важно, что при выполнении ариф-

метических операций объект Series автоматически выравнивает данные, которые проиндексированы в разном порядке:

```
In [29]: obj3
Out[29]:
Ohio     35000
Oregon   16000
Texas    71000
Utah     5000
```

```
In [30]: obj4
Out[30]:
California    NaN
Ohio        35000
Oregon      16000
Texas       71000
```

```
In [31]: obj3 + obj4
Out[31]:
California       NaN
Ohio           70000
Oregon         32000
Texas         142000
Utah             NaN
```

Вопрос о выравнивании данных будет рассмотрен отдельно.

И у самого объекта Series, и у его индекса имеется атрибут `name`, тесно связанный с другими частями функциональности pandas:

```
In [32]: obj4.name = 'population'

In [33]: obj4.index.name = 'state'

In [34]: obj4
Out[34]:
state
California     NaN
Ohio         35000
Oregon       16000
Texas        71000
Name: population
```

Индекс объекта Series можно изменить на месте с помощью присваивания:

```
In [35]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [36]: obj
Out[36]:
Bob     4
Steve   7
Jeff   -5
Ryan    3
```

Объект DataFrame

Объект DataFrame представляет табличную структуру данных, состоящую из упорядоченной коллекции столбцов, причем типы значений (числовой, строковый, булев и т. д.) в разных столбцах могут различаться. В объекте DataFrame хранятся два индекса: по строкам и по столбцам. Можно считать, что это словарь объектов Series. По сравнению с другими похожими на DataFrame структурами,

которые вам могли встречаться раньше (например, `data.frame` в языке R), операции со строками и столбцами в DataFrame в первом приближении симметричны. Внутри объекта данные хранятся в виде одного или нескольких двумерных блоков, а не в виде списка, словаря или еще какой-нибудь коллекции одномерных массивов. Технические детали внутреннего устройства DataFrame выходят за рамки этой книги.

Хотя в DataFrame данные хранятся в двумерном формате, в виде таблицы, нетрудно представить и данные более высокой размерности, если воспользоваться иерархическим индексированием. Эту тему мы обсудим в следующем разделе, она лежит в основе многих продвинутых механизмов обработки данных в pandas.

Есть много способов сконструировать объект DataFrame, один из самых распространенных – на основе словаря списков одинаковой длины или массивов NumPy:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = DataFrame(data)
```

Для получившегося DataFrame автоматически будет построен индекс, как и в случае Series, и столбцы расположатся по порядку:

```
In [38]: frame
Out[38]:
  pop    state    year
0 1.5    Ohio     2000
1 1.7    Ohio     2001
2 3.6    Ohio     2002
3 2.4    Nevada   2001
4 2.9    Nevada   2002
```

Если задать последовательность столбцов, то столбцы DataFrame расположатся строго в указанном порядке:

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
  year  state     pop
0 2000  Ohio      1.5
1 2001  Ohio      1.7
2 2002  Ohio      3.6
3 2001  Nevada    2.4
4 2002  Nevada    2.9
```

Как и в случае Series, если запросить столбец, которого нет в `data`, то он будет заполнен значениями `NaN`:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                    index=['one', 'two', 'three', 'four', 'five'])

In [41]: frame2
```

```
Out[41]:
  year  state     pop   debt
0 2000  Ohio      1.5   NaN
1 2001  Ohio      1.7   NaN
2 2002  Ohio      3.6   NaN
3 2001  Nevada    2.4   NaN
4 2002  Nevada    2.9   NaN

In [42]: frame2.columns
Out[42]: Index([year, state, pop, debt], dtype=object)
```

Столбец DataFrame можно извлечь как объект Series, воспользовавшись нотацией словарей, или с помощью атрибута:

```
In [43]: frame2['state']
Out[43]:
one       Ohio
two       Ohio
three     Ohio
four      Nevada
five      Nevada
Name: state
```

```
In [44]: frame2.year
Out[44]:
one 2000
two 2001
three 2002
four 2001
five 2002
```

Отметим, что возвращенный объект Series имеет тот же индекс, что и DataFrame, а его атрибут `name` установлен соответствующим образом.

Строки также можно извлечь по позиции или по имени, для чего есть два метода, один из них – `ix` с указанием индексного поля (подробнее об этом ниже):

```
In [45]: frame2.ix['three']
Out[45]:
year     2002
state    Ohio
pop       3.6
debt      NaN
Name:   three
```

Столбцы можно модифицировать путем присваивания. Например, пустому столбцу `'debt'` можно было бы присвоить скалярное значение или массив значений:

```
In [46]: frame2['debt'] = 16.5

In [47]: frame2
Out[47]:
      year  state    pop   debt
one   2000  Ohio     1.5   16.5
two   2001  Ohio     1.7   16.5
three 2002  Ohio     3.6   16.5
four  2001  Nevada   2.4   16.5
five  2002  Nevada   2.9   16.5

In [48]: frame2['debt'] = np.arange(5.)
In [49]: frame2
```

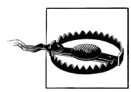
```
Out[49]:
      year  state    pop   debt
one   2000  Ohio     1.5      0
two   2001  Ohio     1.7      1
three 2002  Ohio     3.6      2
four  2001  Nevada   2.4      3
five  2002  Nevada   2.9      4
```

Когда столбцу присваивается список или массив, длина значения должна совпадать с длиной DataFrame. Если же присваивается объект Series, то он будет точно согласован с индексом DataFrame, а в «дырки» будут вставлены значения NA:

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [51]: frame2['debt'] = val

In [52]: frame2
Out[52]:
      year  state    pop   debt
one   2000  Ohio     1.5    NaN
two   2001  Ohio     1.7   -1.2
three 2002  Ohio     3.6    NaN
four  2001  Nevada   2.4   -1.5
five  2002  Nevada   2.9   -1.7
```

Присваивание несуществующему столбцу приводит к созданию нового столбца. Для удаления столбцов служит ключевое слово `del`, как и в обычном словаре:

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'

In [54]: frame2
Out[54]:
      year  state    pop   debt  eastern
one   2000  Ohio     1.5    NaN     True
two   2001  Ohio     1.7   -1.2     True
three 2002  Ohio     3.6    NaN     True
four  2001  Nevada   2.4   -1.5    False
five  2002  Nevada   2.9   -1.7    False

In [55]: del frame2['eastern']

In [56]: frame2.columns
Out[56]: Index([year, state, pop, debt], dtype=object)
```



Столбец, возвращенный в ответ на запрос к DataFrame по индексу, является *представлением*, а не копией данных. Следовательно, любые модификации этого объекта Series, найдут отражение в DataFrame. Чтобы скопировать столбец, нужно вызвать метод `copy` объекта Series.

Еще одна распространенная форма данных – словарь словарей:

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Если передать его конструктору DataFrame, то ключи внешнего словаря будут интерпретированы как столбцы, а ключи внутреннего словаря – как индексы строк:

```
In [58]: frame3 = DataFrame(pop)

In [59]: frame3
Out[59]:
      Nevada  Ohio
2000  NaN      1.5
2001  2.4      1.7
2002  2.9      3.6
```

Разумеется, результат можно транспонировать:

```
In [60]: frame3.T
Out[60]:
        2000  2001  2002
Nevada   NaN   2.4   2.9
Ohio     1.5   1.7   3.6
```

Ключи внутренних словарей объединяются и сортируются для образования индекса результата. Однако этого не происходит, если индекс задан явно:

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])
Out[61]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

Словари объектов Series интерпретируются очень похоже:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [63]: DataFrame(pdata)
Out[63]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
```

Полный перечень возможных аргументов конструктора DataFrame приведен в табл. 5.1.

Если у объектов, возвращаемых при обращении к атрибутам `index` и `columns` объекта DataFrame, установлен атрибут `name`, то он также выводится:

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
```

```
Out[65]:
state  Nevada   Ohio
year
2000      NaN    1.5
2001      2.4    1.7
2002      2.9    3.6
```

Как и в случае Series, атрибут `values` возвращает данные, хранящиеся в DataFrame, в виде двумерного массива ndarray:

```
In [66]: frame3.values
Out[66]:
array([[ nan, 1.5],
       [ 2.4, 1.7],
       [ 2.9, 3.6]])
```

Если у столбцов DataFrame разные типы данных, то dtype массива `values` будет выбран так, чтобы охватить все столбцы:

```
In [67]: frame2.values
Out[67]:
array([[2000, Ohio, 1.5, nan],
       [2001, Ohio, 1.7, -1.2],
       [2002, Ohio, 3.6, nan],
       [2001, Nevada, 2.4, -1.5],
       [2002, Nevada, 2.9, -1.7]], dtype=object)
```

Таблица 5.1. Аргументы конструктора DataFrame

Тип	Примечания
Двумерный ndarray	Матрица данных, дополнительно можно передать метки строк и столбцов
Словарь массивов, списков или кортежей	Каждая последовательность становится столбцом объекта DataFrame. Все последовательности должны быть одинаковой длины
Структурный массив NumPy	Интерпретируется так же, как «словарь массивов»
Словарь объектов Series	Каждое значение становится столбцом. Если индекс явно не задан, то индексы объектов Series объединяются и образуют индекс строк результата
Словарь словарей	Каждый внутренний словарь становится столбцом. Ключи объединяются и образуют индекс строк, как в случае «словаря объектов Series»
Список словарей или объектов Series	Каждый элемент списка становится строкой объекта DataFrame. Объединение ключей словаря или индексов объектов Series становится множеством меток столбцов DataFrame
Список списков или кортежей	Интерпретируется так же, как «двумерный ndarray»
Другой объект DataFrame	Используются индексы DataFrame, если явно не заданы другие индексы

Тип	Примечания
Объект NumPy MaskedArray	Как «двумерный ndarray» с тем отличием, что замаскированные значения становятся отсутствующими в результирующем объекте DataFrame

Индексные объекты

В индексных объектах pandas хранятся метки вдоль осей и прочие метаданные (например, имена осей). Любой массив или иная последовательность меток, указанная при конструировании Series или DataFrame, преобразуется в объект Index:

```
In [68]: obj = Series(range(3), index=['a', 'b', 'c'])

In [69]: index = obj.index

In [70]: index
Out[70]: Index([a, b, c], dtype=object)

In [71]: index[1:]
Out[71]: Index([b, c], dtype=object)
```

Индексные объекты неизменяемы, т. е. пользователь не может их модифицировать:

```
In [72]: index[1] = 'd'
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-72-676fdeb26a68> in <module>()
----> 1 index[1] = 'd'
/Users/wesm/code/pandas/pandas/core/index.pyc in __setitem__(self, key, value)
    302 def __setitem__(self, key, value):
    303 """Disable the setting of values."""
--> 304 raise Exception(str(self.__class__) + ' object is immutable')
    305
    306 def __getitem__(self, key):
Exception: <class 'pandas.core.index.Index'> object is immutable
```

Неизменяемость важна для того, чтобы несколько структур данных могли совместно использовать один и тот же индексный объект, не опасаясь его повредить:

```
In [73]: index = pd.Index(np.arange(3))

In [74]: obj2 = Series([1.5, -2.5, 0], index=index)

In [75]: obj2.index is index
Out[75]: True
```

В табл. 5.2 перечислены включенные в библиотеку индексные классы. При некотором усилии можно даже создать подклассы класса Index, если требуется реализовать специализированную функциональность индексирования вдоль оси.

Многим пользователям подробная информация об индексных объектах не нужна, но они, тем не менее, являются важной частью модели данных pandas.

Таблица 5.2. Основные индексные объекты в pandas

Тип	Примечания
`Index`	Наиболее общий индексный объект, представляющий оси в массиве NumPy, состоящем из объектов Python
`Int64Index`	Специализированный индекс для целых значений
`MultiIndex`	«Иерархический» индекс, представляющий несколько уровней индексирования по одной оси. Можно считать аналогом массива кортежей
`DatetimeIndex`	Хранит временные метки с наносекундной точностью (представлены типом данных NumPy `datetime64`)
`PeriodIndex`	Специализированный индекс для данных о периодах (временных промежутках)

Индексный объект не только похож на массив, но и ведет себя как множество фиксированного размера:

```
In [76]: frame3
Out[76]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [77]: 'Ohio' in frame3.columns
Out[77]: True

In [78]: 2003 in frame3.index
Out[78]: False
```

У любого объекта Index есть ряд свойств и методов для ответа на типичные вопросы о хранящихся в нем данных. Они перечислены в табл. 5.3.

Таблица 5.3. Методы и свойства объекта Index

Метод	Описание
`append`	Конкатенирует с дополнительными индексными объектами, порождая новый объект Index
`diff`	Вычисляет теоретико-множественную разность, представляя ее в виде индексного объекта
`intersection`	Вычисляет теоретико-множественное пересечение

Метод	Описание
union	Вычисляет теоретико-множественное объединение
isin	Вычисляет булев массив, показывающий, содержится ли каждое значение индекса в переданной коллекции
delete	Вычисляет новый индексный объект, получающийся после удаления элемента с индексом i
drop	Вычисляет новый индексный объект, получающийся после удаления переданных значений
insert	Вычисляет новый индексный объект, получающийся после вставки элемента в позицию с индексом i
is_monotonic	Возвращает True, если каждый элемент больше или равен предыдущему
is_unique	Возвращает True, если в индексе нет повторяющихся значений
unique	Вычисляет массив уникальных значений в индексе

Базовая функциональность

В этом разделе мы рассмотрим фундаментальные основы взаимодействия с данными, хранящимися в объектах Series и DataFrame. В последующих главах мы более детально обсудим вопросы анализа и манипуляции данными с применением pandas. Эта книга не задумывалась как исчерпывающая документация по библиотеке pandas, я хотел лишь акцентировать внимание на наиболее важных чертах, оставив не столь употребительные (если не сказать эзотерические) вещи для самостоятельного изучения читателю.

Переиндексация

Для объектов pandas критически важен метод reindex, т. е. возможность создания нового объекта, данные в котором *согласуются* с новым индексом. Рассмотрим простой пример:

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [80]: obj
Out[80]:
d    4.5
b    7.2
a   -5.3
c    3.6
```

Если вызвать reindex для этого объекта Series, то данные будут реорганизованы в соответствии с новым индексом, а если каких-то из имеющихся в этом индексе значений раньше не было, то вместо них будут подставлены отсутствующие значения:

```
In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [82]: obj2
Out[82]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[83]:
a   -5.3
b    7.2
c    3.6
d    4.5
e    0.0
```

Для упорядоченных данных, например временных рядов, иногда желательно произвести интерполяцию, или восполнение отсутствующих значений в процессе переиндексации. Это позволяет сделать параметр `method`; так, если задать для него значение `ffill`, то будет произведено прямое восполнение значений:

```
In [84]: obj3 = Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [85]: obj3.reindex(range(6), method='ffill')
Out[85]:
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
```

В табл. 5.4 перечислены возможные значения параметра `method`. В настоящее время интерполяцию, более сложную, чем прямое и обратное восполнение, нужно производить постфактум.

Таблица 5.4. Значение параметра `method` функции `reindex` (интерполяция)

Значение	Описание
`ffill` или `pad`	Восполнить (или перенести) значения в прямом направлении
`bfill` или `backfill`	Восполнить (или перенести) значения в обратном направлении

В случае объекта DataFrame функция `reindex` может изменять строки, столбцы или то и другое. Если ей передать просто последовательность, то в результирующем объекте переиндексируются строки:

```
In [86]: frame = DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
   ....: columns=['Ohio', 'Texas', 'California'])

In [87]: frame
Out[87]:
```

```
   Ohio  Texas  California
a      0      1           2
c      3      4           5
d      6      7           8

In [88]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
In [89]: fra
    Ohio  Texas  California
a      0      1           2
b    NaN    NaN         NaN
c      3      4           5
d      6      7           8
```

Столбцы можно переиндексировать, задав ключевое слово `columns`:

```
In [90]: states = ['Texas', 'Utah', 'California']

In [91]: frame.reindex(columns=states)
Out[91]:
   Texas  Utah  California
a      1   NaN           2
c      4   NaN           5
d      7   NaN           8
```

Строки и столбцы можно переиндексировать за одну операцию, хотя интерполяция будет применена только к строкам (к оси 0):

```
In [92]: frame.reindex(index=['a', 'b', 'c', 'd'], method='ffill',
    ....: columns=states)
Out[92]:
    Texas  Utah  California
a       1   NaN           2
b       1   NaN           2
c       4   NaN           5
d       7   NaN           8
```

Как мы скоро увидим, переиндексацию можно определить короче путем индексации меток с помощью поля `ix`:

```
In [93]: frame.ix[['a', 'b', 'c', 'd'], states]
Out[93]:
    Texas  Utah  California
a       1   NaN           2
b     NaN   NaN         NaN
c       4   NaN           5
d       7   NaN           8
```

Таблица 5.5. Аргументы функции `reindex`

Аргумент	Описание
`index`	Последовательность, которая должна стать новым индексом. Может быть экземпляром `Index` или любой другой структурой данных Python, похожей на последовательность. Экземпляр `Index` будет использован «как есть», без копирования

Аргумент	Описание
`method`	Метод интерполяции (восполнения), возможные значения приведены в табл. 5.4
`fill_value`	Значение, которой должно подставляться вместо отсутствующих значений, появляющихся в результате переиндексации
`limit`	При прямом или обратном восполнении максимальная длина восполняемой лакуны
`level`	Сопоставить с простым объектом `Index` на указанном уровне `MultiIndex`, иначе выбрать подмножество
`copy`	Не копировать данные, если новый индекс эквивалентен старому. По умолчанию `True` (т. е. всегда копировать данные)

Удаление элементов из оси

Удалить один или несколько элементов из оси просто, если имеется индексный массив или список, не содержащий этих значений. Метод `drop` возвращает новый объект, в котором указанные значения удалены из оси:

```
In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [95]: new_obj = obj.drop('c')

In [96]: new_obj
Out[96]:
a   0
b   1
d   3
e   4

In [97]: obj.drop(['d', 'c'])
Out[97]:
a   0
b   1
e   4
```

В случае DataFrame указанные в индексе значения можно удалить из любой оси:

```
In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),
   ....:                  index=['Ohio', 'Colorado', 'Utah', 'New York'],
   ....:                  columns=['one', 'two', 'three', 'four'])

In [99]: data.drop(['Colorado', 'Ohio'])
Out[99]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15

In [100]: data.drop('two', axis=1)   In [101]: data.drop(['two', 'four'], axis=1)
```

```
Out[100]:
           one  three  four
Ohio         0      2     3
Colorado     4      6     7
Utah         8     10    11
New York    12     14    15
```

```
Out[101]:
           one three
Ohio         0     2
Colorado     4     6
Utah         8    10
New York    12    14
```

Доступ по индексу, выборка и фильтрация

Доступ по индексу к объекту Series (`obj[...]`) работает так же, как для массивов NumPy с тем отличием, что индексами могут быть не только целые, но любые значения из индекса объекта Series. Вот несколько примеров:

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [103]: obj['b']
Out[103]: 1.0

In [104]: obj[1]
Out[104]: 1.0

In [105]: obj[2:4]
Out[105]:
c    2
d    3

In [106]: obj[['b', 'a', 'd']]
Out[106]:
b    1
a    0
d    3

In [107]: obj[[1, 3]]
Out[107]:
b    1
d    3

In [108]: obj[obj < 2]
Out[108]:
a    0
b    1
```

Вырезание с помощью меток отличается от обычного вырезания в Python тем, что конечная точка включается:

```
In [109]: obj['b':'c']
Out[109]:
b    1
c    2
```

Установка с помощью этих методов работает ожидаемым образом:

```
In [110]: obj['b':'c'] = 5

In [111]: obj
Out[111]:
a    0
b    5
c    5
d    3
```

Как мы уже видели, доступ по индексу к DataFrame применяется для извлечения одного или нескольких столбцов путем задания единственного значения или последовательности:

```
In [112]: data = DataFrame(np.arange(16).reshape((4, 4)),
   .....:                  index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                  columns=['one', 'two', 'three', 'four'])

In [113]: data
Out[113]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [114]: data['two']
Out[114]:
Ohio       1
Colorado   5
Utah       9
New York  13
Name: two

In [115]: data[['three', 'one']]
Out[115]:
          three one
Ohio          2   0
Colorado      6   4
Utah         10   8
New York     14  12
```

У доступа по индексу есть несколько частных случаев. Во-первых, выборка строк с помощью вырезания или булева массива:

```
In [116]: data[:2]
Out[116]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [117]: data[data['three'] > 5]
Out[117]:
          one two three four
Colorado    4   5     6    7
Utah        8   9    10   11
New York   12  13    14   15
```

Некоторым читателям такой синтаксис может показаться непоследовательным, но он был выбран исключительно из практических соображений. Еще одна возможность – доступ по индексу с помощью булева DataFrame, например порожденного в результате скалярного сравнения:

```
In [118]: data < 5
Out[118]:
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [119]: data[data < 5] = 0

In [120]: data
Out[120]:
            one    two  three   four
Ohio          0      0      0      0
Colorado      0      5      6      7
Utah          8      9     10     11
New York     12     13     14     15
```

Идея в том, чтобы сделать DataFrame синтаксически более похожим на ndarray в данном частном случае. Для доступа к строкам по индексу с помощью меток я ввел специальное индексное поле `ix`. Оно позволяет выбрать подмножество строк и столбцов DataFrame с применение нотации NumPy, дополненной метками осей. Как я уже говорил, это еще и более лаконичный способ выполнить переиндексацию:

```
In [121]: data.ix['Colorado', ['two', 'three']]
Out[121]:
two     5
three   6
Name: Colorado

In [122]: data.ix[['Colorado', 'Utah'], [3, 0, 1]]
Out[122]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9

In [123]: data.ix[2]
Out[123]:
one      8
two      9
three   10
four    11
Name: Utah

In [124]: data.ix[:'Utah', 'two']
Out[124]:
Ohio      0
Colorado  5
Utah      9
Name: two

In [125]: data.ix[data.three > 5, :3]
Out[125]:
          one two three
Colorado    0   5     6
Utah        8   9    10
New York   12  13    14
```

Таким образом, существует много способов выборки и реорганизации данных, содержащихся в объекте pandas. Для DataFrame краткая сводка многих их них приведена в табл. 5.6. Позже мы увидим, что при работе с иерархическими индексами есть ряд дополнительных возможностей.

Таблица 5.6. Варианты доступа по индексу для объекта DataFrame

Вариант	Примечание
`obj[val]`	Выбрать один столбец или последовательность столбцов из DataFrame. Частные случаи: булев массив (фильтрация строк), срез (вырезание строк) или булев DataFrame (установка значений в позициях, удовлетворяющих некоторому критерию)
`obj.ix[val]`	Выбрать одну строку или подмножество строк из DataFrame
`obj.ix[:, val]`	Выбрать один столбец или подмножество столбцов

Вариант	Примечание
`obj.ix[val1, val2]`	Выбрать строки и столбцы
метод `reindex`	Привести одну или несколько осей в соответствие с новыми индексами
метод `xs`	Выбрать одну строку или столбец по метке и вернуть объект Series
методы `icol, irow`	Выбрать одну строку или столбец соответственно по целочисленному номеру и вернуть объект Series
методы `get_value, set_value`	Выбрать одно значение по меткам строки и столбца

Проектируя pandas, я подспудно ощущал, что нотация `frame[:, col]` для выборки столбца слишком громоздкая (и провоцирующая ошибки), поскольку выборка столбца – одна из самых часто встречающихся операций. Поэтому я пошел на компромисс и перенес все более выразительные операции доступа с помощью индексов-меток в `ix`.

Арифметические операции и выравнивание данных

Одна из самых важных черт pandas – поведение арифметических операций для объектов с разными индексами. Если при сложении двух объектов обнаруживаются различные пары индексов, то результирующий индекс будет объединением индексов. Рассмотрим простой пример:

```
In [126]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [127]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])

In [128]: s1
Out[128]:
a    7.3
c   -2.5
d    3.4
e    1.5
```

```
In [129]: s2
Out[129]:
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
```

Сложение этих объектов дает:

```
In [130]: s1 + s2
Out[130]:
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
```

Вследствие внутреннего выравнивания данных образуются отсутствующие значения в позициях, для которых не нашлось соответственной пары. Отсутствующие значения распространяются на последующие операции.

В случае DataFrame выравнивание производится как для строк, так и для столбцов:

```
In [131]: df1 = DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                 index=['Ohio', 'Texas', 'Colorado'])

In [132]: df2 = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                 index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
In [133]: df1
Out[133]:
          b  c  d
Ohio      0  1  2
Texas     3  4  5
Colorado  6  7  8
```

```
In [134]: df2
Out[134]:
          b   d   e
Utah      0   1   2
Ohio      3   4   5
Texas     6   7   8
Oregon    9  10  11
```

При сложении этих объектов получается DataFrame, индекс и столбцы которого являются объединениями индексов и столбцов слагаемых:

```
In [135]: df1 + df2
Out[135]:
            b    c    d    e
Colorado  NaN  NaN  NaN  NaN
Ohio        3  NaN    6  NaN
Oregon    NaN  NaN  NaN  NaN
Texas       9  NaN   12  NaN
Utah      NaN  NaN  NaN  NaN
```

Восполнение значений в арифметических методах

При выполнении арифметических операций с объектами, проиндексированными по-разному, иногда желательно поместить специальное значение, например 0, в позиции операнда, которым в другом операнде соответствует отсутствующая позиция:

```
In [136]: df1 = DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))

In [137]: df2 = DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
```

```
In [138]: df1
Out[138]:
    a   b   c   d
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
```

```
In [139]: df2
Out[139]:
    a   b   c   d   e
0   0   1   2   3   4
1   5   6   7   8   9
2  10  11  12  13  14
3  15  16  17  18  19
```

Сложение этих объектов порождает отсутствующие значения в позициях, которые имеются не в обоих операндах:

```
In [140]: df1 + df2
Out[140]:
     a    b    c    d   e
0    0    2    4    6  NaN
1    9   11   13   15  NaN
2   18   20   22   24  NaN
3  NaN  NaN  NaN  NaN  NaN
```

Теперь я вызову метод `add` объекта `df1` и передам ему объект `df2` и значение параметра `fill_value`:

```
In [141]: df1.add(df2, fill_value=0)
Out[141]:
     a   b   c   d   e
0    0   2   4   6   4
1    9  11  13  15   9
2   18  20  22  24  14
3   15  16  17  18  19
```

Точно так же, выполняя переиндексацию объекта Series или DataFrame, можно указать восполняемое значение:

```
In [142]: df1.reindex(columns=df2.columns, fill_value=0)
Out[142]:
    a   b   c   d   e
0   0   1   2   3   0
1   4   5   6   7   0
2   8   9  10  11   0
```

Таблица 5.7. Гибкие арифметические методы

Метод	Описание
`add`	Сложение (+)
`sub`	Вычитание (-)
`div`	Деление (/)
`mul`	Умножение (*)

Операции между DataFrame и Series

Как и в случае массивов NumPy, арифметические операции между DataFrame и Series корректно определены. В качестве пояснительного примера рассмотрим вычисление разности между двумерным массивом и одной из его строк:

```
In [143]: arr = np.arange(12.).reshape((3, 4))

In [144]: arr
Out[144]:

array([[ 0., 1., 2., 3.],
```

```
       [ 4., 5., 6., 7.],
       [ 8., 9., 10., 11.]])

In [145]: arr[0]
Out[145]: array([ 0., 1., 2., 3.])

In [146]: arr - arr[0]
Out[146]:
array([[ 0., 0., 0., 0.],
       [ 4., 4., 4., 4.],
       [ 8., 8., 8., 8.]])
```

Это называется *укладыванием* и подробно объясняется в главе 12. Операции между DataFrame и Series аналогичны:

```
In [147]: frame = DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [148]: series = frame.ix[0]

In [149]: frame
Out[149]:
         b   d   e
Utah     0   1   2
Ohio     3   4   5
Texas    6   7   8
Oregon   9  10  11

In [150]: series
Out[150]:
b   0
d   1
e   2
Name: Utah
```

По умолчанию при выполнении арифметических операций между DataFrame и Series индекс Series сопоставляется со столбцами DataFrame, а укладываются строки:

```
In [151]: frame - series
Out[151]:
        b   d   e
Utah    0   0   0
Ohio    3   3   3
Texas   6   6   6
Oregon  9   9   9
```

Если какой-нибудь индекс не найден либо в столбцах DataFrame, либо в индексе Series, то объекты переиндексируются для образования объединения:

```
In [152]: series2 = Series(range(3), index=['b', 'e', 'f'])

In [153]: frame + series2
Out[153]:
        b   d   e   f
Utah    0 NaN   3 NaN
Ohio    3 NaN   6 NaN
Texas   6 NaN   9 NaN
Oregon  9 NaN  12 NaN
```

Если вы хотите вместо этого сопоставлять строки, а укладывать столбцы, то должны будете воспользоваться каким-нибудь арифметическим методом. Например:

```
In [154]: series3 = frame['d']

In [155]: frame
Out[155]:
        b   d   e
Utah    0   1   2
Ohio    3   4   5
Texas   6   7   8
Oregon  9  10  11

In [156]: series3
Out[156]:
Utah     1
Ohio     4
Texas    7
Oregon  10
Name: d

In [157]: frame.sub(series3, axis=0)
Out[157]:
         b   d   e
Utah    -1   0   1
Ohio    -1   0   1
Texas   -1   0   1
Oregon  -1   0   1
```

Передаваемый номер оси – это ось, *вдоль которой производится сопоставление*. В данном случае мы хотим сопоставлять с индексом строк DataFrame и укладывать поперек.

Применение функций и отображение

Универсальные функции NumPy (поэлементные методы массивов) отлично работают и с объектами pandas:

```
In [158]: frame = DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....: index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [159]: frame
Out[159]:
               b         d         e
Utah   -0.204708  0.478943 -0.519439
Ohio   -0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189 -1.296221

In [160]: np.abs(frame)
Out[160]:
               b         d         e
Utah    0.204708  0.478943  0.519439
Ohio    0.555730  1.965781  1.393406
Texas   0.092908  0.281746  0.769023
Oregon  1.246435  1.007189  1.296221
```

Еще одна часто встречающаяся операция – применение функции, определенной для одномерных массивов, к каждому столбцу или строке. Именно это и делает метод `apply` объекта DataFrame:

```
In [161]: f = lambda x: x.max() - x.min()

In [162]: frame.apply(f)
Out[162]:
b   1.802165
d   1.684034

In [163]: frame.apply(f, axis=1)
Out[163]:
Utah    0.998382
Ohio    2.521511
```

```
e   2.689627                    Texas   0.676115
                                Oregon  2.542656
```

Многие из наиболее распространенных статистик массивов (например, `sum` и `mean`) – методы DataFrame, поэтому применять `apply` в этом случае необязательно. Функция, передаваемая методу `apply`, не обязана возвращать скалярное значение, она может вернуть и объект Series, содержащий несколько значений:

```
In [164]: def f(x):
    .....: return Series([x.min(), x.max()], index=['min', 'max'])

In [165]: frame.apply(f)
Out[165]:
              b          d          e
min   -0.555730   0.281746  -1.296221
max    1.246435   1.965781   1.393406
```

Можно использовать и поэлементные функции Python. Допустим, требуется вычислить форматированную строку для каждого элемента `frame` с плавающей точкой. Это позволяет сделать метод `applymap`:

```
In [166]: format = lambda x: '%.2f' % x

In [167]: frame.applymap(format)
Out[167]:
           b     d      e
Utah   -0.20  0.48  -0.52
Ohio   -0.56  1.97   1.39
Texas   0.09  0.28   0.77
Oregon  1.25  1.01  -1.30
```

Этот метод называется `applymap`, потому что в классе Series есть метод `map` для применения функции к каждому элементу:

```
In [168]: frame['e'].map(format)
Out[168]:
Utah     -0.52
Ohio      1.39
Texas     0.77
Oregon   -1.30
Name: e
```

Сортировка и ранжирование

Сортировка набора данных по некоторому критерию – еще одна важная встроенная операция. Для лексикографической сортировки по индексу служит метод `sort_index`, который возвращает новый отсортированный объект:

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])

In [170]: obj.sort_index()
Out[170]:
```

```
a   1
b   2
c   3
d   0
```

В случае DataFrame можно сортировать по индексу, ассоциированному с любой осью:

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],
   .....: columns=['d', 'a', 'b', 'c'])

In [172]: frame.sort_index()
Out[172]:
        d  a  b  c
one     4  5  6  7
three   0  1  2  3

In [173]: frame.sort_index(axis=1)
Out[173]:
        a  b  c  d
three   1  2  3  0
one     5  6  7  4
```

По умолчанию данные сортируются в порядке возрастания, но можно отсортировать их и в порядке убывания:

```
In [174]: frame.sort_index(axis=1, ascending=False)
Out[174]:
        d  c  b  a
three   0  3  2  1
one     4  7  6  5
```

Для сортировки Series по значениям служит метод `order`:

```
In [175]: obj = Series([4, 7, -3, 2])

In [176]: obj.order()
Out[176]:
2    -3
3     2
0     4
1     7
```

Отсутствующие значения по умолчанию оказываются в конце Series:

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])

In [178]: obj.order()
Out[178]:
4    -3
5     2
0     4
2     7
1   NaN
3   NaN
```

Объект DataFrame можно сортировать по значениям в одном или нескольких столбцах. Для этого имена столбцов следует передать в качестве значения параметра `by`:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [180]: frame
Out[180]:
   a   b
0  0   4
1  1   7
2  0  -3
3  1   2

In [181]: frame.sort_index(by='b')
Out[181]:
   a   b
2  0  -3
3  1   2
0  0   4
1  1   7
```

Для сортировки по нескольким столбцам следует передать список имен:

```
In [182]: frame.sort_index(by=['a', 'b'])
Out[182]:
   a   b
2  0  -3
0  0   4
3  1   2
1  1   7
```

Ранжирование тесно связано с сортировкой, заключается оно в присваивании рангов – от единицы до числа присутствующих в массиве элементов. Это аналогично косвенным индексам, порождаемым методом `numpy.argsort`, с тем отличием, что существует правило обработки связанных рангов. Для ранжирования применяется метод `rank` объектов Series и DataFrame; по умолчанию `rank` обрабатывает связанные ранги, присваивая каждой группе средний ранг:

```
In [183]: obj = Series([7, -5, 7, 4, 2, 0, 4])

In [184]: obj.rank()
Out[184]:
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
```

Ранги можно также присваивать в соответствии с порядком появления в данных:

```
In [185]: obj.rank(method='first')
Out[185]:
0    6
1    1
2    7
3    4
4    3
5    2
6    5
```

Естественно, можно ранжировать и в порядке убывания:

```
In [186]: obj.rank(ascending=False, method='max')
Out[186]:
0    2
1    7
2    2
3    4
4    5
5    6
6    4
```

В табл. 5.8 приведен перечень способов обработки связанных рангов. DataFrame умеет вычислять ранги как по строкам, так и по столбцам:

```
In [187]: frame = DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....: 'c': [-2, 5, 8, -2.5]})

In [188]: frame
Out[188]:
   a     b     c
0  0   4.3  -2.0
1  1   7.0   5.0
2  0  -3.0   8.0
3  1   2.0  -2.5

In [189]: frame.rank(axis=1)
Out[189]:
   a  b  c
0  2  3  1
1  1  3  2
2  2  1  3
3  2  3  1
```

Таблица 5.8. Способы обработки связанных рангов

Способ	Описание
`'average'`	По умолчанию: одинаковым значениям присвоить средний ранг
`'min'`	Всем элементам группы присвоить минимальный ранг
`'max'`	Всем элементам группы присвоить максимальный ранг
`'first'`	Присваивать ранги в порядке появления значений в наборе данных

Индексы по осям с повторяющимися значениями

Во всех рассмотренных до сих пор примерах метки на осях (значения индекса) были уникальны. Хотя для многих функций pandas (например, `reindex`) требуется уникальность меток, в общем случае это необязательно. Рассмотрим небольшой объект Series с повторяющимися значениями в индексе:

```
In [190]: obj = Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [191]: obj
Out[191]:
a   0
a   1
b   2
b   3
c   4
```

О том, являются значения уникальными или нет, можно узнать, опросив свойство `is_unique`:

```
In [192]: obj.index.is_unique
Out[192]: False
```

Выборка данных – одна из основных операций, поведение которых меняется в зависимости от наличия или отсутствия дубликатов. При доступе по индексу, встречающемуся несколько раз, возвращается объект Series; если же индекс одиночный, то скалярное значение:

```
In [193]: obj['a']          In [194]: obj['c']
Out[193]:                   Out[194]: 4
a   0
a   1
```

Такое же правило действует и для доступа к строкам в DataFrame:

```
In [195]: df = DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [196]: df
Out[196]:
          0         1         2
a  0.274992   0.228913   1.352917
a  0.886429  -2.001637  -0.371843
b  1.669025  -0.438570  -0.539741
b  0.476985   3.248944  -1.021228

In [197]: df.ix['b']
Out[197]:
          0         1         2
b  1.669025  -0.438570  -0.539741
b  0.476985   3.248944  -1.021228
```

Редукция и вычисление описательных статистик

Объекты pandas оснащены набором стандартных математических и статистических методов. Большая их часть попадает в категорию *редукций*, или *сводных статистик* – методов, которые вычисляют единственное значение (например, сумму или среднее) для Series или объект Series – для строк либо столбцов DataFrame. По сравнению с эквивалентными методами массивов NumPy, все они игнорируют отсутствующие значения. Рассмотрим небольшой объект DataFrame:

```
In [198]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                 [np.nan, np.nan], [0.75, -1.3]],
   .....:                index=['a', 'b', 'c', 'd'],
   .....:                columns=['one', 'two'])

In [199]: df
```

```
Out[199]:
     one    two
a   1.40    NaN
b   7.10   -4.5
c    NaN    NaN
d   0.75   -1.3
```

Метод `sum` объекта DataFrame возвращает Series, содержащий суммы по столбцам:

```
In [200]: df.sum()
Out[200]:
one    9.25
two   -5.80
```

Если передать параметр `axis=1`, то суммирование будет производиться по строкам:

```
In [201]: df.sum(axis=1)
Out[201]:
a    1.40
b    2.60
c     NaN
d   -0.55
```

Отсутствующие значения исключаются, если только не является отсутствующим весь срез (в данном случае строка или столбец). Это можно подавить, задав параметр `skipna`:

```
In [202]: df.mean(axis=1, skipna=False)
Out[202]:
a      NaN
b    1.300
c      NaN
d   -0.275
```

Перечень часто употребляемых параметров методов редукции приведен в табл. 5.9.

Таблица 5.9. Параметры методов редукции

Метод	Описание
`axis`	Ось, по которой производится редуцирование. В случае DataFrame 0 означает строки, 1 – столбцы.
`skipna`	Исключать отсутствующие значения. По умолчанию `True`
`level`	Редуцировать с группировкой по уровням, если индекс по оси иерархический (MultiIndex)

Некоторые методы, например `idxmin` и `idxmax`, возвращают косвенные статистики, скажем, индекс, при котором достигается минимум или максимум:

```
In [203]: df.idxmax()
Out[203]:
one   b
two   d
```

Есть также *аккумулирующие* методы:

```
In [204]: df.cumsum()
Out[204]:
     one   two
a   1.40   NaN
b   8.50  -4.5
c    NaN   NaN
d   9.25  -5.8
```

Наконец, существуют методы, не относящиеся ни к редуцирующим, ни к аккумулирующим. Примером может служить метод `describe`, который возвращает несколько сводных статистик за одно обращение:

```
In [205]: df.describe()
Out[205]:
              one         two
count    3.000000    2.000000
mean     3.083333   -2.900000
std      3.493685    2.262742
min      0.750000   -4.500000
25%      1.075000   -3.700000
50%      1.400000   -2.900000
75%      4.250000   -2.100000
max      7.100000   -1.300000
```

В случае нечисловых данных `describe` возвращает другие сводные статистики:

```
In [206]: obj = Series(['a', 'a', 'b', 'c'] * 4)

In [207]: obj.describe()
Out[207]:
count     16
unique     3
top        a
freq       8
```

Полный список сводных статистик и родственных методов приведен в табл. 5.10.

Таблица 5.10. Описательные и сводные статистики

Метод	Описание
`count`	Количество значений, исключая отсутствующие
`describe`	Вычисляет набор сводных статистик для Series или для каждого столбца DataFrame
`min, max`	Вычисляет минимальное или максимальное значение

Метод	Описание
argmin, argmax	Вычисляет позицию в индексе (целые числа), при котором достигается минимальное или максимальное значение соответственно
idxmin, idxmax	Вычисляет значение индекса, при котором достигается минимальное или максимальное значение соответственно
quantile	Вычисляет выборочный квантиль в диапазоне от 0 до 1
sum	Сумма значений
mean	Среднее значение
median	Медиана (50%-ый квантиль)
mad	Среднее абсолютное отклонение от среднего
var	Выборочная дисперсия
std	Выборочное стандартное отклонение
skew	Асимметрия (третий момент)
kurt	Куртозис (четвертый момент)
cumsum	Нарастающая сумма
cummin, cummax	Нарастающий минимум или максимум соответственно
cumprod	Нарастающее произведение
diff	Первая арифметическая разность (полезно для временных рядов)
pct_change	Вычисляет процентное изменение

Корреляция и ковариация

Некоторые сводные статистики, например корреляция и ковариация, вычисляются по парам аргументов. Рассмотрим объекты DataFrame, содержащие цены акций и объемы биржевых сделок, взятые с сайта Yahoo! Finance:

```
import pandas.io.data as web

all_data = {}
for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']:
  all_data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')

price = DataFrame({tic: data['Adj Close']
              for tic, data in all_data.iteritems()})

volume = DataFrame({tic: data['Volume']
              for tic, data in all_data.iteritems()})
```

Теперь вычислим процентные изменения цен:

```
In [209]: returns = price.pct_change()

In [210]: returns.tail()
```

```
Out[210]:
                AAPL        GOOG         IBM        MSFT
Date
2009-12-24   0.034339    0.011117    0.004420    0.002747
2009-12-28   0.012294    0.007098    0.013282    0.005479
2009-12-29  -0.011861   -0.005571   -0.003474    0.006812
2009-12-30   0.012147    0.005376    0.005468   -0.013532
2009-12-31  -0.004300   -0.004416   -0.012609   -0.015432
```

Метод `corr` объекта Series вычисляет корреляцию перекрывающихся, отличных от NA, выровненных по индексу значений в двух объектах Series. Соответственно, метод `cov` вычисляет ковариацию:

```
In [211]: returns.MSFT.corr(returns.IBM)
Out[211]: 0.49609291822168838

In [212]: returns.MSFT.cov(returns.IBM)
Out[212]: 0.00021600332437329015
```

С другой стороны, методы `corr` и `cov` объекта DataFrame возвращают соответственно полную корреляционную или ковариационную матрицу в виде DataFrame:

```
In [213]: returns.corr()
Out[213]:
           AAPL       GOOG        IBM       MSFT
AAPL   1.000000   0.470660   0.410648   0.424550
GOOG   0.470660   1.000000   0.390692   0.443334
IBM    0.410648   0.390692   1.000000   0.496093
MSFT   0.424550   0.443334   0.496093   1.000000

In [214]: returns.cov()
Out[214]:
           AAPL       GOOG        IBM       MSFT
AAPL   0.001028   0.000303   0.000252   0.000309
GOOG   0.000303   0.000580   0.000142   0.000205
IBM    0.000252   0.000142   0.000367   0.000216
MSFT   0.000309   0.000205   0.000216   0.000516
```

С помощью метода `corrwith` объекта DataFrame можно вычислить попарные корреляции между столбцами или строками DataFrame и другим объектом Series или DataFrame. Если передать ему объект Series, то будет возвращен Series, содержащий значения корреляции, вычисленные для каждого столбца:

```
In [215]: returns.corrwith(returns.IBM)
Out[215]:
AAPL   0.410648
GOOG   0.390692
IBM    1.000000
MSFT   0.496093
```

Если передать объект DataFrame, то будут вычислены корреляции столбцов с соответственными именами. Ниже я вычисляю корреляции процентных изменений с объемом сделок:

```
In [216]: returns.corrwith(volume)
Out[216]:
AAPL   -0.057461
GOOG    0.062644
IBM    -0.007900
MSFT   -0.014175
```

Если передать `axis=1`, то будут вычислены корреляции строк. Во всех случаях перед началом вычислений данные выравниваются по меткам.

Уникальные значения, счетчики значений и членство

Еще один класс методов служит для извлечения информации о значениях, хранящихся в одномерном объекте Series. Для иллюстрации рассмотрим пример:

```
In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

Метод `unique` возвращает массив уникальных значений в Series:

```
In [218]: uniques = obj.unique()

In [219]: uniques
Out[219]: array([c, a, d, b], dtype=object)
```

Уникальные значения необязательно возвращаются в отсортированном порядке, но могут быть отсортированы впоследствии, если это необходимо (`uniques.sort()`). Метод `value_counts` вычисляет объект Series, содержащий частоты встречаемости значений:

```
In [220]: obj.value_counts()
Out[220]:
c   3
a   3
b   2
d   1
```

Для удобства этот объект отсортирован по значениям в порядке убывания. Функция `value_counts` может быть также вызвана как метод pandas верхнего уровня и в таком случае применима к любому массиву или последовательности:

```
In [221]: pd.value_counts(obj.values, sort=False)
Out[221]:
a   3
b   2
c   3
d   1
```

Наконец, метод `isin` вычисляет булев вектор членства в множестве и может быть полезен для фильтрации набора данных относительно подмножества значений в объекте Series или столбце DataFrame:

```
In [222]: mask = obj.isin(['b', 'c'])

In [223]: mask
Out[223]:
0    True
1   False
2   False
3   False
4   False
5    True
6    True
7    True
8    True

In [224]: obj[mask]
Out[224]:
0   c
5   b
6   b
7   c
8   c
```

Справочная информация по этим методам приведена в табл. 5.11.

Таблица 5.11. Уникальные значения, счетчики значений и методы «раскладывания»

Метод	Описание
`isin`	Вычисляет булев массив, показывающий, содержится ли каждое принадлежащее Series значение в переданной последовательности
`unique`	Вычисляет массив уникальных значений в Series и возвращает их в порядке появления
`value_counts`	Возвращает объект Series, который содержит уникальное значение в качестве индекса и его частоту в качестве соответствующего значения. Отсортирован в порядке убывания частот

Иногда требуется вычислить гистограмму нескольких взаимосвязанных столбцов в DataFrame. Приведем пример:

```
In [225]: data = DataFrame({'Qu1': [1, 3, 4, 3, 4],
    .....: 'Qu2': [2, 3, 1, 2, 3],
    .....: 'Qu3': [1, 5, 2, 4, 4]})

In [226]: data
Out[226]:
   Qu1  Qu2  Qu3
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Передача `pandas.value_counts` методу `apply` этого объекта DataFrame дает:

```
In [227]: result = data.apply(pd.value_counts).fillna(0)

In [228]: result
Out[228]:
   Qu1  Qu2  Qu3
1    1    1    1
2    0    2    1
3    2    2    0
```

```
4    2    0    2
5    0    0    1
```

Обработка отсутствующих данных

Отсутствующие данные – типичное явление в большинстве аналитических приложений. При проектировании pandas в качестве одной из целей ставилась задача сделать работу с отсутствующими данными как можно менее болезненной. Например, выше мы видели, что при вычислении всех описательных статистик для объектов pandas отсутствующие данные не учитываются.

В pandas для представления отсутствующих данных в любых массивах – как чисел с плавающей точкой, так и иных – используется значение с плавающей точкой `NaN` (не число). Это просто *признак*, который легко распознать:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [230]: string_data
Out[230]:
0    aardvark
1   artichoke
2         NaN
3     avocado

In [231]: string_data.isnull()
Out[231]:
0   False
1   False
2   True
3   False
```

Встроенное в Python значение `None` также рассматривается как отсутствующее в массивах объектов:

```
In [232]: string_data[0] = None

In [233]: string_data.isnull()
Out[233]:
0   True
1   False
2   True
3   False
```

Я не утверждаю, что представление отсутствующих значений в pandas оптимально, но оно простое и в разумной степени последовательное. Принимая во внимание характеристики производительности и простой API, это наилучшее решение, которое я смог придумать в отсутствие истинного типа данных NA или выделенной комбинации бит среди типов данных NumPy. Поскольку разработка NumPy продолжается, в будущем ситуация может измениться.

Таблица 5.12. Методы обработки отсутствующих данных

Аргумент	Описание
`dropna`	Фильтрует метки оси в зависимости от того, существуют ли для метки отсутствующие данные, причем есть возможность указать различные пороги, определяющие, какое количество отсутствующих данных считать допустимым

Аргумент	Описание
fillna	Восполняет отсутствующие данные указанным значением или использует какой-нибудь метод интерполяции, например 'ffill' или 'bfill'
isnull	Возвращает объект, содержащий булевы значения, которые показывают, какие значения отсутствуют
notnull	Логическое отрицание isnull

Фильтрация отсутствующих данных

Существует ряд средств для фильтрации отсутствующих данных. Конечно, можно сделать это и вручную, но часто бывает полезен метод dropna. Для Series он возвращает другой объект Series, содержащий только данные и значения индекса, отличные от NA:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
0   1.0
2   3.5
4   7.0
```

Естественно, это можно было бы вычислить и самостоятельно с помощью булевой индексации:

```
In [237]: data[data.notnull()]
Out[237]:
0   1.0
2   3.5
4   7.0
```

В случае объектов DataFrame все немного сложнее. Можно отбрасывать строки или столбцы, если они содержат только NA-значения или хотя бы одно NA-значение. По умолчанию метод dropna отбрасывает все строки, содержащие хотя бы одно отсутствующее значение:

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
   .....:                   [NA, NA, NA], [NA, 6.5, 3.]])

In [239]: cleaned = data.dropna()

In [240]: data                In [241]: cleaned
Out[240]:                     Out[241]:
     0    1    2                   0    1    2
0    1  6.5    3             0     1  6.5    3
1    1  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5    3
```

Если передать параметр `how='all'`, то будут отброшены строки, которые целиком состоят из отсутствующих значений:

```
In [242]: data.dropna(how='all')
Out[242]:
     0    1    2
0    1  6.5    3
1    1  NaN  NaN
3  NaN  6.5    3
```

Для отбрасывания столбцов достаточно передать параметр `axis=1`:

```
In [243]: data[4] = NA

In [244]: data
Out[244]:
     0    1    2    4
0    1  6.5    3  NaN
1    1  NaN  NaN  NaN
2  NaN  NaN  NaN NaN
3  NaN  6.5    3 NaN
```

```
In [245]: data.dropna(axis=1, how='all')
Out[245]:
     0    1    2
0    1  6.5    3
1    1  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5    3
```

Родственный способ фильтрации строк DataFrame в основном применяется к временным рядам. Допустим, требуется оставить только строки, содержащие определенное количество наблюдений. Этот порог можно задать с помощью аргумента `thresh`:

```
In [246]: df = DataFrame(np.random.randn(7, 3))

In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA

In [248]: df
Out[248]:
          0         1         2
0 -0.577087       NaN       NaN
1  0.523772       NaN       NaN
2 -0.713544       NaN       NaN
3 -1.860761       NaN  0.560145
4 -1.265934       NaN -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

```
In [249]: df.dropna(thresh=3)
Out[249]:
          0         1         2
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Восполнение отсутствующих данных

Иногда отсутствующие данные желательно не отфильтровывать (и потенциально вместе с ними отбрасывать полезные данные), а каким-то способом заполнить «дыры». В большинстве случаев для этой цели можно использовать метод `fillna`. Ему передается константа, подставляемая вместо отсутствующих значений:

```
In [250]: df.fillna(0)
Out[250]:
```

```
          0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Если передать методу `fillna` словарь, то можно будет подставлять вместо отсутствующих данных значение, зависящее от столбца:

```
In [251]: df.fillna({1: 0.5, 3: -1})
Out[251]:
          0         1         2
0 -0.577087  0.500000       NaN
1  0.523772  0.500000       NaN
2 -0.713544  0.500000       NaN
3 -1.860761  0.500000  0.560145
4 -1.265934  0.500000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Метод `fillna` возвращает новый объект, но можно также модифицировать существующий объект на месте:

```
# всегда возвращает ссылку на заполненный объект
In [252]: _ = df.fillna(0, inplace=True)

In [253]: df
Out[253]:
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Те же методы интерполяции, что применяются для переиндексации, годятся и для `fillna`:

```
In [254]: df = DataFrame(np.random.randn(6, 3))

In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA

In [256]: df
Out[256]:
0 1 2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674       NaN  1.004812
3  1.327195       NaN -1.549106
4  0.022185       NaN       NaN
```

```
5   0.862580        NaN        NaN

In [257]: df.fillna(method='ffill')
Out[257]:
          0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185  1.349742 -1.549106
5  0.862580  1.349742 -1.549106

In [258]: df.fillna(method='ffill', limit=2)
Out[258]:
          0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674  1.349742  1.004812
3  1.327195  1.349742 -1.549106
4  0.022185       NaN -1.549106
5  0.862580       NaN -1.549106
```

При некоторой изобретательности можно использовать `fillna` и другими способами. Например, можно передать среднее или медиану объекта Series:

```
In [259]: data = Series([1., NA, 3.5, NA, 7])

In [260]: data.fillna(data.mean())
Out[260]:
0   1.000000
1   3.833333
2   3.500000
3   3.833333
4   7.000000
```

Справочная информация о методе `fillna` приведена в табл. 5.13.

Таблица 5.13. Аргументы метода `fillna`

Аргумент	Описание
`value`	Скалярное значение или похожий на словарь объект для восполнения отсутствующих значений
`method`	Метод интерполяции. По умолчанию, если не задано других аргументов, предполагается метод `'ffill'`
`axis`	Ось, по которой производится восполнение. По умолчанию `axis=0`
`inplace`	Модифицировать исходный объект, не создавая копию
`limit`	Для прямого и обратного восполнения максимальное количество непрерывных заполняемых промежутков

Иерархическое индексирование

Иерархическое индексирование – важная особенность pandas, позволяющая организовать несколько (два и более) *уровней* индексирования по одной оси. Говоря абстрактно, это способ работать с многомерными данными, представив их в форме с меньшей размерностью. Начнем с простого примера – создадим объект Series с индексом в виде списка списков или массивов:

```
In [261]: data = Series(np.random.randn(10),
   .....: index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
```

```
    .....:        [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])

In [262]: data
Out[262]:
a  1    0.670216
   2    0.852965
   3   -0.955869
b  1   -0.023493
   2   -2.304234
   3   -0.652469
c  1   -1.218302
   2   -1.332610
d  2    1.074623
   3    0.723642
```

Здесь мы видим отформатированное представление Series с мультииндексом (`MultiIndex`). «Разрывы» в представлении индекса означают «взять значение вышестоящей метки».

```
In [263]: data.index
Out[263]:
MultiIndex
[('a', 1) ('a', 2) ('a', 3) ('b', 1) ('b', 2) ('b', 3) ('c', 1)
 ('c', 2) ('d', 2) ('d', 3)]
```

Для иерархически индексированного объекта возможен доступ по так называемому *частичному* индексу, что позволяет лаконично записывать выборку подмножества данных:

```
In [264]: data['b']
Out[264]:
1   -0.023493
2   -2.304234
3   -0.652469

In [265]: data['b':'c']
Out[265]:
b  1  -0.023493
   2  -2.304234
   3  -0.652469
c  1  -1.218302
   2  -1.332610

In [266]: data.ix[['b', 'd']]
Out[266]:
b  1  -0.023493
   2  -2.304234
   3  -0.652469
d  2   1.074623
   3   0.723642
```

В некоторых случаях возможна даже выборка с «внутреннего» уровня:

```
In [267]: data[:, 2]
Out[267]:
a   0.852965
b  -2.304234
c  -1.332610
d   1.074623
```

Иерархическое индексирование играет важнейшую роль в изменении формы данных и групповых операциях, в том числе построении сводных таблиц. Напри-

мер, эти данные можно было бы преобразовать в DataFrame с помощью метода `unstack`:

```
In [268]: data.unstack()
Out[268]:
1 2 3
a   0.670216   0.852965  -0.955869
b  -0.023493  -2.304234  -0.652469
c  -1.218302  -1.332610        NaN
d        NaN   1.074623   0.723642
```

Обратной к `unstack` операцией является `stack`:

```
In [269]: data.unstack().stack()
Out[269]:
a  1   0.670216
   2   0.852965
   3  -0.955869
b  1  -0.023493
   2  -2.304234
   3  -0.652469
c  1  -1.218302
   2  -1.332610
d  2   1.074623
   3   0.723642
```

Методы `stack` и `unstack` будут подробно рассмотрены в главе 7.

В случае DataFrame иерархический индекс можно построить по любой оси:

```
In [270]: frame = DataFrame(np.arange(12).reshape((4, 3)),
   .....:                   index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
   .....:                   columns=[['Ohio', 'Ohio', 'Colorado'],
   .....:                            ['Green', 'Red', 'Green']])
In [271]: frame
Out[271]:
       Ohio        Colorado
      Green   Red     Green
a  1      0     1         2
   2      3     4         5
b  1      6     7         8
   2      9    10        11
```

Уровни иерархии могут иметь имена (как строки или любые объекты Python). В таком случае они будут показаны при выводе на консоль (не путайте имена индексов с метками на осях!):

```
In [272]: frame.index.names = ['key1', 'key2']

In [273]: frame.columns.names = ['state', 'color']

In [274]: frame
Out[274]:
state    Ohio        Colorado
```

```
color   Green   Red     Green
key1 key2
a  1         0     1         2
   2         3     4         5
b  1         6     7         8
   2         9    10        11
```

Доступ по частичному индексу, как и раньше, позволяет выбирать группы столбцов:

```
In [275]: frame['Ohio']
Out[275]:
color   Green  Red
key1 key2
a  1         0    1
   2         3    4
b  1         6    7
   2         9   10
```

Мультииндекс можно создать отдельно, а затем использовать повторно; в показанном выше объекте DataFrame столбцы с именами уровней можно было бы создать так:

```
tiIndex.from_arrays([['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']],
                    names=['state', 'color'])
```

Уровни переупорядочения и сортировки

Иногда требуется изменить порядок уровней на оси или отсортировать данные по значениям на одной уровне. Метод `swaplevel` принимает номера или имена двух уровней и возвращает новый объект, в котором эти уровни переставлены (но во всех остальных отношениях данные не изменяются):

```
In [276]: frame.swaplevel('key1', 'key2')
Out[276]:
state    Ohio          Colorado
color   Green   Red      Green
key2 key1
1    a      0     1          2
2    a      3     4          5
1    b      6     7          8
2    b      9    10         11
```

С другой стороны, метод `sortlevel` выполняет устойчивую сортировку данных, используя только значения на одном уровне. После перестановки уровней обычно вызывают также `sortlevel`, чтобы лексикографически отсортировать результат:

```
In [277]: frame.sortlevel(1)
Out[277]:
state    Ohio        Colorado
```

```
In [278]: frame.swaplevel(0, 1).sortlevel(0)
 Out[278]:
state    Ohio         Colorado
```

```
color   Green   Red     Green   color   Green   Red     Green
key1 key2                       key2 key1
a    1      0     1         2   1    a      0     1         2
b    1      6     7         8        b      6     7         8
a    2      3     4         5   2    a      3     4         5
b    2      9    10        11        b      9    10        11
```

Производительность выборки данных из иерархически индексированных объектов будет гораздо выше, если индекс отсортирован лексикографически, начиная с самого внешнего уровня, т. е. в результате вызова `sortlevel(0)` или `sort_index()`.

Сводная статистика по уровню

У многих методов объектов DataFrame и Series, вычисляющих сводные и описательные статистики, имеется параметр `level` для задания уровня, на котором требуется производить суммирование по конкретной оси. Рассмотрим тот же объект DataFrame, что и выше; мы можем суммировать по уровню для строк или для столбцов:

```
In [279]: frame.sum(level='key2')
Out[279]:
state    Ohio         Colorado
color   Green   Red      Green
key2
1           6     8         10
2          12    14         16

In [280]: frame.sum(level='color', axis=1)
Out[280]:
color    Green    Red
key1 key2
a    1       2      1
     2       8      4
b    1      14      7
     2      20     10
```

Реализовано это с помощью имеющегося в pandas механизма `groupby`, который мы подробно рассмотрим позже.

Работа со столбцами DataFrame

Не так уж редко возникает необходимость использовать один или несколько столбцов DataFrame в качестве индекса строк; альтернативно можно переместить индекс строк в столбцы DataFrame. Рассмотрим пример:

```
In [281]: frame = DataFrame({'a': range(7), 'b': range(7, 0, -1),
   .....:                    'c': ['one', 'one', 'one', 'two', 'two', 'two', 'two'],
   .....:                    'd': [0, 1, 2, 0, 1, 2, 3]})

In [282]: frame
```

```
Out[282]:
   a  b    c  d
0  0  7  one  0
1  1  6  one  1
2  2  5  one  2
3  3  4  two  0
4  4  3  two  1
5  5  2  two  2
6  6  1  two  3
```

Метод `set_index` объекта DataFrame создает новый DataFrame, используя в качестве индекса один или несколько столбцов:

```
In [283]: frame2 = frame.set_index(['c', 'd'])

In [284]: frame2
Out[284]:
      a b
c   d
one 0 0 7
    1 1 6
    2 2 5
two 0 3 4
    1 4 3
    2 5 2
    3 6 1
```

По умолчанию столбцы удаляются из DataFrame, хотя их можно и оставить:

```
In [285]: frame.set_index(['c', 'd'], drop=False)
Out[285]:
      a b   c d
c   d
one 0 0 7 one 0
    1 1 6 one 1
    2 2 5 one 2
two 0 3 4 two 0
    1 4 3 two 1
    2 5 2 two 2
    3 6 1 two 3
```

Есть также метод `reset_index`, который делает прямо противоположное `set_index`; уровни иерархического индекса перемещаются в столбцы:

```
In [286]: frame2.reset_index()
Out[286]:
    c d a b
0 one 0 0 7
1 one 1 1 6
2 one 2 2 5
3 two 0 3 4
4 two 1 4 3
5 two 2 5 2
6 two 3 6 1
```

Другие возможности pandas

Ниже перечислено еще несколько возможностей, которые могут пригодиться вам в экспериментах с данными.

Доступ по целочисленному индексу

При работе с объектами pandas, проиндексированными целыми числами, начинающие часто попадают в ловушку из-за некоторых различий с семантикой доступа по индексу к встроенным в Python структурам данных, в частности, спискам и кортежам. Например, вряд ли вы ожидаете ошибки в таком коде:

```
ser = Series(np.arange(3.))
ser[-1]
```

В данном случае pandas могла бы прибегнуть к целочисленному индексированию, но я не знаю никакого общего и безопасного способа сделать это, не внося тонких ошибок. Здесь мы имеем индекс, содержащий значения 0, 1, 2, но автоматически понять, чего хочет пользователь (индекс по метке или по позиции) трудно:

```
In [288]: ser
Out[288]:
0   0
1   1
2   2
```

С другой стороны, когда индекс не является целым числом, неоднозначности не возникает:

```
In [289]: ser2 = Series(np.arange(3.), index=['a', 'b', 'c'])

In [290]: ser2[-1]
Out[290]: 2.0
```

Чтобы не оставлять места разноречивым интерпретациям, принято решение: если индекс по оси содержит индексаторы, то доступ к данным по целочисленному индексу всегда трактуется как доступ по метке. Это относится и к полю `ix`:

```
In [291]: ser.ix[:1]
Out[291]:
0   0
1   1
```

В случае, когда требуется надежный доступ по номеру позиции вне зависимости от типа индекса, можно использовать метод `iget_value` объекта Series или методы `irow` и `icol` объекта DataFrame:

```
In [292]: ser3 = Series(range(3), index=[-5, 1, 3])

In [293]: ser3.iget_value(2)
```

```
Out[293]: 2

In [294]: frame = DataFrame(np.arange(6).reshape(3, 2)), index=[2, 0, 1])

In [295]: frame.irow(0)
Out[295]:
0   0
1   1
Name: 2
```

Структура данных Panel

Хотя структура данных Panel и не является основной темой этой книги, она существует в pandas и может рассматриваться как трехмерный аналог DataFrame. При разработке pandas основное внимание уделялось манипуляциям с табличными данными, поскольку о них проще рассуждать, а наличие иерархических индексов в большинстве случаев позволяет обойтись без настоящих N-мерных массивов.

Для создания Panel используется словарь объектов DataFrame или трехмерный массив ndarray:

```
import pandas.io.data as web

pdata = pd.Panel(dict((stk, web.get_data_yahoo(stk, '1/1/2009', '6/1/2012'))
                      for stk in ['AAPL', 'GOOG', 'MSFT', 'DELL']))
```

Каждый элемент Panel (аналог столбца в DataFrame) является объектом DataFrame:

```
In [297]: pdata
Out[297]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 861 (major) x 6 (minor)
Items: AAPL to MSFT
Major axis: 2009-01-02 00:00:00 to 2012-06-01 00:00:00
Minor axis: Open to Adj Close

In [298]: pdata = pdata.swapaxes('items', 'minor')

In [299]: pdata['Adj Close']
Out[299]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 861 entries, 2009-01-02 00:00:00 to 2012-06-01 00:00:00
Data columns:
AAPL   861   non-null values
DELL   861   non-null values
GOOG   861   non-null values
MSFT   861   non-null values
dtypes: float64(4)
```

Основанное на поле `ix` индексирование по меткам обобщается на три измерения, поэтому мы можем выбрать все данные за конкретную дату или диапазон дат следующим образом:

```
In [300]: pdata.ix[:, '6/1/2012', :]
Out[300]:
Open     High     Low   Close  Volume       Adj   Close
AAPL   569.16  572.65  560.52  560.99  18606700  560.99
DELL    12.15   12.30   12.05   12.07  19396700   12.07
GOOG   571.79  572.65  568.35  570.98   3057900  570.98
MSFT    28.76   28.96   28.44   28.45  56634300   28.45

In [301]: pdata.ix['Adj Close', '5/22/2012':, :]
Out[301]:
              AAPL   DELL    GOOG   MSFT
Date
2012-05-22  556.97  15.08  600.80  29.76
2012-05-23  570.56  12.49  609.46  29.11
2012-05-24  565.32  12.45  603.66  29.07
2012-05-25  562.29  12.46  591.53  29.06
2012-05-29  572.27  12.66  594.34  29.56
2012-05-30  579.17  12.56  588.23  29.34
2012-05-31  577.73  12.33  580.86  29.19
2012-06-01  560.99  12.07  570.98  28.45
```

Альтернативный способ представления панельных данных, особенно удобный для подгонки статистических моделей, – в виде «стопки» объектов DataFrame:

```
In [302]: stacked = pdata.ix[:, '5/30/2012':, :].to_frame()

In [303]: stacked
Out[303]:
Open High Low Close Volume Adj Close
major minor
2012-05-30 AAPL  569.20  579.99  566.56  579.17  18908200  579.17
           DELL   12.59   12.70   12.46   12.56  19787800   12.56
           GOOG  588.16  591.90  583.53  588.23   1906700  588.23
           MSFT   29.35   29.48   29.12   29.34  41585500   29.34
2012-05-31 AAPL  580.74  581.50  571.46  577.73  17559800  577.73
           DELL   12.53   12.54   12.33   12.33  19955500   12.33
           GOOG  588.72  590.00  579.00  580.86   2968300  580.86
           MSFT   29.30   29.42   28.94   29.19  39134000   29.19
2012-06-01 AAPL  569.16  572.65  560.52  560.99  18606700  560.99
           DELL   12.15   12.30   12.05   12.07  19396700   12.07
           GOOG  571.79  572.65  568.35  570.98   3057900  570.98
           MSFT   28.76   28.96   28.44   28.45  56634300   28.45
```

У объекта DataFrame есть метод `to_panel` – обращение `to_frame`:

```
In [304]: stacked.to_panel()
Out[304]:
<class 'pandas.core.panel.Panel'>
Dimensions: 6 (items) x 3 (major) x 4 (minor)
Items: Open to Adj Close
Major axis: 2012-05-30 00:00:00 to 2012-06-01 00:00:00
Minor axis: AAPL to MSFT
```

ГЛАВА 6. Чтение и запись данных, форматы файлов

Описанные в этой книге инструменты были бы бесполезны, если бы в программе на Python не было возможности легко импортировать и экспортировать данные Я буду рассматривать в основном ввод и вывод с помощью объектов pandas, хотя, разумеется, в других библиотеках нет недостатка в соответствующих средствах. Например, в NumPy имеются низкоуровневые, но очень быстрые функции для загрузки и сохранения данных, включая и поддержку файлов, спроецированных на память. Подробнее об этом см. главу 12.

Обычно средства ввода-вывода относят к нескольким категориям: чтение файлов в текстовом или каком-то более эффективном двоичном формате, загрузка из баз данных и взаимодействие с сетевыми источниками, например API доступа к веб.

Чтение и запись данных в текстовом формате

Python превратился в излюбленный язык манипулирования текстом и файлами благодаря простому синтаксису взаимодействия с файлами, интуитивно понятным структурам данных и таким удобным средствам, как упаковка и распаковка кортежей.

В библиотеке pandas имеется ряд функций для чтения табличных данных, представленных в виде объекта DataFrame. Все они перечислены в табл. 6.1, хотя чаще всего вы будете иметь дело с функциями `read_csv` и `read_table`.

Таблица 6.1. Функции чтения в pandas

Функция	Описание
`read_csv`	Загружает данные с разделителями из файла, URL-адреса или похожего на файл объекта. По умолчанию разделителем является запятая
`read_table`	Загружает данные с разделителями из файла, URL-адреса или похожего на файл объекта. По умолчанию разделителем является символ табуляции (`'\t'`)

Функция	Описание
read_fwf	Читает данные в формате с фиксированной шириной столбцов (без разделителей)
read_clipboard	Вариант read_table, который читает данные из буфера обмена. Полезно для преобразования в таблицу данных на веб-странице

Я дам краткий обзор этих функций, которые служат для преобразования текстовых данных в объект DataFrame. Их параметры можно отнести к нескольким категориям:

- *индексирование*: какие столбцы рассматривать как индекс возвращаемого DataFrame и откуда брать имена столбцов: из файла, от пользователя или вообще ниоткуда;
- *выведение типа и преобразование данных*: включает определенные пользователем преобразования значений и список маркеров отсутствующих данных;
- *разбор даты и времени*: включает средства комбинирования, в том числе сбор данных о дате и времени из нескольких исходных столбцов в один результирующий;
- *итерирование*: поддержка обхода очень больших файлов;
- *проблемы «грязных» данных*: пропуск заголовка или концевика, комментариев и другие мелочи, например, обработка числовых данных, в которых тройки разрядов разделены запятыми.

Выведение типа – одна из самых важных черт этих функций; это означает, что пользователю необязательно явно задавать, содержат столбцы данные с плавающей точкой, целочисленные, булевы или строковые. Правда, для обработки дат и нестандартных типов требуется больше усилий. Начнем с текстового файла, содержащего короткий список данных через запятую (формат CSV):

```
In [846]: !cat ch06/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Поскольку данные разделены запятыми, мы можем прочитать их в DataFrame с помощью функции read_csv:

```
In [847]: df = pd.read_csv('ch06/ex1.csv')

In [848]: df
Out[848]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Можно было бы также воспользоваться функцией read_table, указав разделитель:

```
In [849]: pd.read_table('ch06/ex1.csv', sep=',')
Out[849]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Я здесь пользуюсь командой Unix `cat`, которая печатает содержимое файла на экране без какого-либо форматирования. Если вы работаете в Windows, можете с тем же успехом использовать команду `type`.

В файле не всегда есть строка-заголовок. Рассмотрим такой файл:

```
In [850]: !cat ch06/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

Прочитать его можно двумя способами. Можно поручить pandas выбор имен столбцов по умолчанию, а можно задать их самостоятельно:

```
In [851]: pd.read_csv('ch06/ex2.csv', header=None)
Out[851]:
   X.1  X.2  X.3  X.4    X.5
0    1    2    3    4  hello
1    5    6    7    8  world
2    9   10   11   12    foo

In [852]: pd.read_csv('ch06/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[852]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Допустим, мы хотим, чтобы столбец `message` стал индексом возвращаемого объекта DataFrame. Этого можно добиться, задав аргумент `index_col`, в котором указать, что индексом будет столбец с номером 4 или с именем `'message'`:

```
In [853]: names = ['a', 'b', 'c', 'd', 'message']

In [854]: pd.read_csv('ch06/ex2.csv', names=names, index_col='message')
Out[854]:
         a   b   c   d
message
hello    1   2   3   4
world    5   6   7   8
foo      9  10  11  12
```

Если вы хотите сформировать иерархический индекс из нескольких столбцов, то просто передайте список их номеров или имен:

```
In [855]: !cat ch06/csv_mindex.csv
key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [856]: parsed = pd.read_csv('ch06/csv_mindex.csv', index_col=['key1', 'key2'])
In [857]: parsed
Out[857]:
          value1  value2
key1 key2
one  a         1       2
     b         3       4
     c         5       6
     d         7       8
two  a         9      10
     b        11      12
     c        13      14
     d        15      16
```

Иногда в таблице нет фиксированного разделителя, а для разделения полей используются пробелы или еще какой-то символ. В таком случае можно передать функции `read_table` регулярное выражение вместо разделителя. Рассмотрим такой текстовый файл:

```
In [858]: list(open('ch06/ex3.txt'))
Out[858]:
['            A         B         C\n',
 'aaa -0.264438 -1.026059 -0.619500\n',
 'bbb  0.927272  0.302904 -0.032399\n',
 'ccc -0.264273 -0.386314 -0.217601\n',
 'ddd -0.871858 -0.348382  1.100491\n']
```

В данном случае поля разделены переменным числом пробелов и, хотя можно было бы переформатировать данные вручную, проще передать регулярное выражение `\s+`:

```
In [859]: result = pd.read_table('ch06/ex3.txt', sep='\s+')

In [860]: result
Out[860]:
            A         B         C
aaa -0.264438 -1.026059 -0.619500
bbb  0.927272  0.302904 -0.032399
ccc -0.264273 -0.386314 -0.217601
ddd -0.871858 -0.348382  1.100491
```

Поскольку имен столбцов на одно меньше, чем число строк, `read_table` делает вывод, что в данном частном случае первый столбец должен быть индексом DataFrame.

У функций разбора много дополнительных аргументов, которые помогают справиться с широким разнообразием файловых форматов (см. табл. 6.2). Например, параметр `skiprows` позволяет пропустить первую, третью и четвертую строку файла:

```
In [861]: !cat ch06/ex4.csv
# привет!
a,b,c,d,message
# хотелось немного усложнить тебе жизнь
# а нечего читать CSV-файла на компьютере
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [862]: pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])
Out[862]:
   a   b   c   d   message
0  1   2   3   4     hello
1  5   6   7   8     world
2  9  10  11  12       foo
```

Обработка отсутствующих значений – важная и зачастую сопровождаемая тонкими нюансами часть разбора файла. Отсутствующие значения обычно либо вообще опущены (пустые строки), либо представлены специальными *маркерами*. По умолчанию в pandas используется набор общеупотребительных маркеров: `NA`, `-1.#IND` и `NULL`:

```
In [863]: !cat ch06/ex5.csv
something,a,b,c,d,message
one,1,2,3,4,NA
two,5,6,,8,world
three,9,10,11,12,foo

In [864]: result = pd.read_csv('ch06/ex5.csv')

In [865]: result
Out[865]:
  something   a   b    c   d   message
0       one   1   2    3   4       NaN
1       two   5   6  NaN   8     world
2     three   9  10   11  12       foo

In [866]: pd.isnull(result)
Out[866]:
   something      a      b      c      d  message
0      False  False  False  False  False     True
1      False  False  False   True  False    False
2      False  False  False  False  False    False
```

Параметр `na_values` может принимать список или множество строк, рассматриваемых как маркеры отсутствующих значений:

```
In [867]: result = pd.read_csv('ch06/ex5.csv', na_values=['NULL'])

In [868]: result

Out[868]:
   something   a   b    c   d   message
0        one   1   2    3   4       NaN
1        two   5   6  NaN   8     world
2      three   9  10   11  12       foo
```

Если в разных столбцах применяются разные маркеры, то их можно задать с помощью словаря:

```
In [869]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [870]: pd.read_csv('ch06/ex5.csv', na_values=sentinels)
Out[870]:
   something   a   b    c   d   message
0        one   1   2    3   4       NaN
1        NaN   5   6  NaN   8     world
2      three   9  10   11  12       NaN
```

Таблица 6.2. Аргументы функций read_csv и read_table

Аргумент	Описание
`path`	Строка, обозначающая путь в файловой системе, URL-адрес или похожий на файл объект
`sep` или `delimiter`	Последовательность символов или регулярное выражение, служащее для разделения полей в строке
`header`	Номер строки, содержащей имена столбцов. По умолчанию равен 0 (первая строка). Если строки-заголовка нет, должен быть равен `None`
`index_col`	Номера или имена столбцов, трактуемых как индекс строк в результирующем объекте. Может быть задан один номер (имя) или список номеров (имен), определяющий иерархический индекс
`names`	Список имен столбцов результирующего объекта, задается, если `header=None`
`skiprows`	Количество игнорируемых начальных строк или список номеров игнорируемых строк (нумерация начинается с 0)
`na_values`	Последовательность значений, интерпретируемых как маркеры отсутствующих данных
`comment`	Один или несколько символов, начинающих комментарий, который продолжается до конца строки
`parse_dates`	Пытаться разобрать данные как дату и время; по умолчанию `False`. Если равен `True`, то производится попытка разобрать все столбцы. Можно также задать список столбцов, которые следует объединить перед разбором (если, например, время и даты заданы в разных столбцах)
`keep_date_col`	В случае, когда для разбора данных столбцы объединяются, следует ли отбрасывать объединенные столбцы. По умолчанию `True`

Аргумент	Описание
converters	Словарь, содержащий отображение номеров или имен столбцов на функции. Например, {'foo': f} означает, что нужно применить функцию f ко всем значением в столбце foo
dayfirst	При разборе потенциально неоднозначных дат предполагать международный формат (т. е. 7/6/2012 означает «7 июня 2012»). По умолчанию False
date_parser	Функция, применяемая для разбора дат
nrows	Количество читаемых строк от начала файла
iterator	Возвращает объект TextParser для чтения файла порциями
chunksize	Размер порции при итерировании
skip_footer	Сколько строк в конце файла игнорировать
verbose	Печатать разного рода информацию о ходе разбора, например, количество отсутствующих значений, помещенных в нечисловые столбцы
encoding	Кодировка текста в случае Unicode. Например, 'utf-8' означает, что текст представлен в кодировке UTF-8
squeeze	Если в результате разбора данных оказалось, что имеется только один столбец, вернуть объект Series
thousands	Разделитель тысяч, например, ',' или '.'

Чтение текстовых файлов порциями

Для обработки очень больших файлов или для того чтобы определить правильный набор аргументов, необходимых для обработки большого файла, иногда требуется прочитать небольшой фрагмент файла или последовательно читать файл небольшими порциями .

```
In [871]: result = pd.read_csv('ch06/ex6.csv')

In [872]: result
Out[872]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns:
one     10000  non-null values
two     10000  non-null values
three   10000  non-null values
four    10000  non-null values
key     10000  non-null values
dtypes: float64(4), object(1)
```

Чтобы прочитать только небольшое число строк (а не весь файл), нужно задать это число в параметре nrows:

```
In [873]: pd.read_csv('ch06/ex6.csv', nrows=5)
Out[873]:
```

```
        one       two     three      four key
0  0.467976 -0.038649 -0.295344 -1.824726   L
1 -0.358893  1.404453  0.704965 -0.200638   B
2 -0.501840  0.659254 -0.421691 -0.057688   G
3  0.204886  1.074134  1.388361 -0.982404   R
4  0.354628 -0.133116  0.283763 -0.837063   Q
```

Для чтения файла порциями задайте с помощью параметра `chunksize` размер порции в строках:

```
In [874]: chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)

In [875]: chunker
Out[875]: <pandas.io.parsers.TextParser at 0x8398150>
```

Объект `TextParser`, возвращаемый функцией `read_csv`, позволяет читать файл порциями размера `chunksize`. Например, можно таким образом итеративно читать файл `ex6.csv`, агрегируя счетчики значений в столбце `'key'`:

```
chunker = pd.read_csv('ch06/ex6.csv', chunksize=1000)

tot = Series([])
for piece in chunker:
 tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.order(ascending=False)
```

Имеем:

```
In [877]: tot[:10]
Out[877]:
E   368
X   364
L   346
O   343
Q   340
M   338
J   337
F   335
K   334
H   330
```

У объекта `TextParser` имеется также метод `get_chunk`, который позволяет читать куски произвольного размера.

Вывод данных в текстовом формате

Данные можно экспортировать в формате с разделителями. Рассмотрим одну из приведенных выше операций чтения CSV-файла:

```
In [878]: data = pd.read_csv('ch06/ex5.csv')

In [879]: data
```

```
Out[879]:
   something   a   b    c   d   message
0        one   1   2    3   4       NaN
1        two   5   6  NaN   8     world
2      three   9  10   11  12       foo
```

С помощью метода `to_csv` объекта DataFrame мы можем вывести данные в файл через запятую:

```
In [880]: data.to_csv('ch06/out.csv')

In [881]: !cat ch06/out.csv
,something,a,b,c,d,message
0,one,1,2,3.0,4,
1,two,5,6,,8,world
2,three,9,10,11.0,12,foo
```

Конечно, допустимы и другие разделители (при выводе в `sys.stdout` результат отправляется на стандартный вывод, обычно на экран):

```
In [882]: data.to_csv(sys.stdout, sep='|')
|something|a|b|c|d|message
0|one|1|2|3.0|4|
1|two|5|6||8|world
2|three|9|10|11.0|12|foo
```

Отсутствующие значения представлены пустыми строками. Но можно вместо этого указать какой-нибудь маркер:

```
In [883]: data.to_csv(sys.stdout, na_rep='NULL')
,something,a,b,c,d,message
0,one,1,2,3.0,4,NULL
1,two,5,6,NULL,8,world
2,three,9,10,11.0,12,foo
```

Если не указано противное, выводятся метки строк и столбцов. Но и те, и другие можно подавить:

```
In [884]: data.to_csv(sys.stdout, index=False, header=False)
one,1,2,3.0,4,
two,5,6,,8,world
three,9,10,11.0,12,foo
```

Можно также вывести лишь подмножество столбцов, задав их порядок:

```
In [885]: data.to_csv(sys.stdout, index=False, cols=['a', 'b', 'c'])
a,b,c
1,2,3.0
5,6,
9,10,11.0
```

У объекта Series также имеется метод `to_csv`:

```
In [886]: dates = pd.date_range('1/1/2000', periods=7)

In [887]: ts = Series(np.arange(7), index=dates)

In [888]: ts.to_csv('ch06/tseries.csv')

In [889]: !cat ch06/tseries.csv
2000-01-01 00:00:00,0
2000-01-02 00:00:00,1
2000-01-03 00:00:00,2
2000-01-04 00:00:00,3
2000-01-05 00:00:00,4
2000-01-06 00:00:00,5
2000-01-07 00:00:00,6
```

Подходящим образом задав параметры (заголовок отсутствует, первый столбец считается индексом), CSV-представление объекта Series можно прочитать методом `read_csv`, но существует вспомогательный метод `from_csv`, который позволяет сделать это немного проще:

```
In [890]: Series.from_csv('ch06/tseries.csv', parse_dates=True)
Out[890]:
2000-01-01   0
2000-01-02   1
2000-01-03   2
2000-01-04   3
2000-01-05   4
2000-01-06   5
2000-01-07   6
```

Дополнительные сведения о методах `to_csv` и `from_csv` смотрите в строках документации в IPython.

Ручная обработка данных в формате с разделителями

Как правило, табличные данные можно загрузить с диска с помощью функции `pandas.read_table` и родственных ей. Но иногда требуется ручная обработка. Не так уж необычно встретить файл, в котором одна или несколько строк сформированы неправильно, что сбивает `read_table`. Для иллюстрации базовых средств рассмотрим небольшой CSV-файл:

```
In [891]: !cat ch06/ex7.csv
"a","b","c"
"1","2","3"
"1","2","3","4"
```

Для любого файла с односимвольным разделителем можно воспользоваться стандартным модулем Python `csv`. Для этого передайте открытый файл или объект, похожий на файл, методу `csv.reader`:

```
import csv
f = open('ch06/ex7.csv')

reader = csv.reader(f)
```

Итерирование файла с помощью объекта reader дает кортежи значений в каждой строке после удаления кавычек:

```
In [893]: for line in reader:
   .....:     print line
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3', '4']
```

Далее можно произвести любые манипуляции, необходимые для преобразования данных к нужному виду. Например:

```
In [894]: lines = list(csv.reader(open('ch06/ex7.csv')))

In [895]: header, values = lines[0], lines[1:]

In [896]: data_dict = {h: v for h, v in zip(header, zip(*values))}

In [897]: data_dict
Out[897]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

Встречаются различные вариации CSV-файлов. Для определения нового формата со своим разделителем, соглашением об употреблении кавычек и способе завершения строк необходимо определить простой подкласс класса `csv.Dialect`:

```
class my_dialect(csv.Dialect):
  lineterminator = '\n'
  delimiter = ';'
  quotechar = '"'

reader = csv.reader(f, dialect=my_dialect)
```

Параметры диалекта CSV можно задать также в виде именованных параметров `csv.reader`, не определяя подкласса:

```
reader = csv.reader(f, delimiter='|')
```

Возможные атрибуты `csv.Dialect` вместе с назначением каждого описаны в табл. 6.3.

Таблица 6.3. Параметры диалекта CSV

Аргумент	Описание
`delimiter`	Односимвольная строка, определяющая разделитель полей. По умолчанию `','`
`lineterminator`	Завершитель строк при выводе, по умолчанию `'\r\n'`. Объект reader игнорирует этот параметр, используя вместо него платформенное соглашение о концах строк

Аргумент	Описание
quotechar	Символ закавычивания для полей, содержащих специальные символы (например, разделитель). По умолчанию '"'
quoting	Соглашение об употреблении кавычек. Допустимые значения: csv.QUOTE_ALL (заключать в кавычки все поля), csv.QUOTE_MINIMAL (только поля, содержащие специальные символы, например разделитель), csv.QUOTE_NONNUMERIC и csv.QUOTE_NON (не заключать в кавычки). Полное описание см. в документации. По умолчанию QUOTE_MINIMAL
skipinitialspace	Игнорировать пробелы после каждого разделителя. По умолчанию False
doublequote	Как обрабатывать символ кавычки внутри поля. Если True, добавляется второй символ кавычки. Полное описание поведения см. в документации.
escapechar	Строка для экранирования разделителя в случае, когда quoting равно csv.QUOTE_NONE. По умолчанию экранирование выключено.

Если в файле употребляются более сложные или фиксированные многосимвольные разделители, то воспользоваться модулем csv не удастся. В таких случаях придется разбивать строку на части и производить другие действия по очистке данных, применяя метод строки split или метод регулярного выражения re.split.

Для *записи* файлов с разделителями вручную можно использовать метод csv.writer. Он принимает объект, который представляет открытый, допускающий запись файл, и те же параметры диалекта и форматирования, что csv.reader:

```
with open('mydata.csv', 'w') as f:
  writer = csv.writer(f, dialect=my_dialect)
  writer.writerow(('one', 'two', 'three'))
  writer.writerow(('1', '2', '3'))
  writer.writerow(('4', '5', '6'))
  writer.writerow(('7', '8', '9'))
```

Данные в формате JSON

Формат JSON (JavaScript Object Notation) стал очень популярен для обмена данными по протоколу HTTP между веб-сервером и браузером или другим клиентским приложением. Этот формат обладает куда большей гибкостью, чем табличный текстовый формат типа CSV. Например:

```
obj = """
{"name": "Wes",
 "places_lived": ["United States", "Spain", "Germany"],
 "pet": null,
 "siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
              {"name": "Katie", "age": 33, "pet": "Cisco"}]
```

```
}
"""
```

Данные в формате JSON очень напоминают код на Python с тем отличием, что отсутствующее значение обозначается `null`, и еще некоторыми нюансами (например, запрещается ставить запятую после последнего элемента списка). Базовыми типами являются объекты (словари), массивы (списки), строки, числа, булевы значения и null. Ключ любого объекта должен быть строкой. На Python существует несколько библиотек для чтения и записи JSON-данных. Здесь я воспользуюсь модулем `json`, потому что он входит в стандартную библиотеку Python. Для преобразования JSON-строки в объект Python служит метод `json.loads`:

```
In [899]: import json

In [900]: result = json.loads(obj)

In [901]: result
Out[901]:
{u'name': u'Wes',
 u'pet': None,
 u'places_lived': [u'United States', u'Spain', u'Germany'],
 u'siblings': [{u'age': 25, u'name': u'Scott', u'pet': u'Zuko'},
               {u'age': 33, u'name': u'Katie', u'pet': u'Cisco'}]}
```

Напротив, метод `json.dumps` преобразует объект Python в формат JSON:

```
In [902]: asjson = json.dumps(result)
```

Как именно преобразовывать объект JSON или список таких объектов в DataFrame или еще какую-то структуру данных для анализа, решать вам. Для удобства предлагается возможность передать список объектов JSON конструктору DataFrame и выбрать подмножество полей данных:

```
In [903]: siblings = DataFrame(result['siblings'], columns=['name', 'age'])

In [904]: siblings
Out[904]:
   name   age
0  Scott   25
1  Katie   33
```

Более полный пример чтения и манипулирования данными в формате JSON (включая и вложенные записи) приведен при рассмотрении базы данных о продуктах питания USDA в следующей главе.

Сейчас идет работа по добавлению в pandas быстрых написанных на C средств для экспорта в формате JSON (`to_json`) и декодирования данных в этом формате (`from_json`). На момент написания этой книги они еще не были готовы.

XML и HTML: разбор веб-страниц

На Python написано много библиотек для чтения и записи данных в вездесущих форматах HTML и XML. В частности, библиотека lxml (*http://lxml.de*) известна высокой производительностью при разборе очень больших файлов. Для lxml имеется несколько программных интерфейсов; сначала я продемонстрирую интерфейс `lxml.html` для работы с HTML, а затем разберу XML-документ с помощью `lxml.objectify`.

Многие сайты показывают данные в виде HTML-таблиц, удобных для просмотра в браузере, но не предлагают их в таких машиночитаемых форматах, как JSON или XML. Так, например, обстоит дело с данными об биржевых опционах на сайте Yahoo! Finance. Для тех, кто не в курсе, скажу, что опцион – это производный финансовый инструмент (дериватив), который дает право покупать (опцион на покупку, или *колл-опцион*) или продавать (опцион на продажу, или *пут-опцион*) акции компании по некоторой цене (*цене исполнения*) в промежутке времени между текущим моментом и некоторым фиксированным моментом в будущем (*конечной датой*). Колл- и пут-опционы торгуются с разными ценами исполнения и конечными датами; эти данные можно найти в таблицах на сайте Yahoo! Finance.

Для начала решите, с какого URL-адреса вы хотите загружать данные, затем откройте его с помощью средств из библиотеки `urllib2` и разберите поток, пользуясь `lxml`:

```
from lxml.html import parse
from urllib2 import urlopen

parsed = parse(urlopen('http://finance.yahoo.com/q/op?s=AAPL+Options'))

doc = parsed.getroot()
```

Имея этот объект, мы можем выбрать все HTML-теги указанного типа, например, теги `table`, внутри которых находятся интересующие нас данные. Для примера получим список всех гиперссылок в документе, они представляются в HTML тегом `a`. Вызовем метод `findall` корневого элемента документа, передав ему выражение XPath (это язык, на котором записываются «запросы» к документу):

```
In [906]: links = doc.findall('.//a')

In [907]: links[15:20]
Out[907]:
[<Element a at 0x6c488f0>,
 <Element a at 0x6c48950>,
 <Element a at 0x6c489b0>,
 <Element a at 0x6c48a10>,
 <Element a at 0x6c48a70>]
```

Но это объекты, представляющие HTML-элементы; чтобы получить URL и текст ссылки, нам нужно воспользоваться методом `get` элемента (для получения URL) или методом `text_content` (для получения текста):

```
In [908]: lnk = links[28]

In [909]: lnk
Out[909]: <Element a at 0x6c48dd0>

In [910]: lnk.get('href')
Out[910]: 'http://biz.yahoo.com/special.html'

In [911]: lnk.text_content()
Out[911]: 'Special Editions'
```

Таким образом, получение всех гиперссылок в документе сводится к списковому включению:

```
In [912]: urls = [lnk.get('href') for lnk in doc.findall('.//a')]

In [913]: urls[-10:]
Out[913]:
['http://info.yahoo.com/privacy/us/yahoo/finance/details.html',
 'http://info.yahoo.com/relevantads/',
 'http://docs.yahoo.com/info/terms/',
 'http://docs.yahoo.com/info/copyright/copyright.html',
 'http://help.yahoo.com/l/us/yahoo/finance/forms_index.html',
 'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
 'http://help.yahoo.com/l/us/yahoo/finance/quotes/fitadelay.html',
 'http://www.capitaliq.com',
 'http://www.csidata.com',
 'http://www.morningstar.com/']
```

Что касается отыскания нужных таблиц в документе, то это делается методом проб и ошибок; на некоторых сайтах решение этой задачи упрощается, потому что таблица имеет атрибут `id`. Я нашел, какие таблицы содержат данные о колл- и пут-опционах:

```
tables = doc.findall('.//table')
calls = tables[9]
puts = tables[13]
```

В каждой таблице имеется строка-заголовок, а за ней идут строки с данными:

```
In [915]: rows = calls.findall('.//tr')
```

Для всех строк, включая заголовок, мы хотим извлечь текст из каждой ячейки; в случае заголовка ячейками являются элементы `th`, а для строк данных – элементы `td`:

```
def _unpack(row, kind='td'):
  elts = row.findall('.//%s' % kind)
  return [val.text_content() for val in elts]
```

Таким образом, получаем:

```
In [917]: _unpack(rows[0], kind='th')
Out[917]: ['Strike', 'Symbol', 'Last', 'Chg', 'Bid', 'Ask', 'Vol', 'Open Int']

In [918]: _unpack(rows[1], kind='td')
```

```
Out[918]:
['295.00',
 'AAPL120818C00295000',
 '310.40',
 ' 0.00',
 '289.80',
 '290.80',
 '1',
 '169']
```

Теперь для преобразования данных в объект DataFrame осталось объединить все описанные шаги вместе. Поскольку числовые данные по-прежнему записаны в виде строк, возможно, потребуется преобразовать некоторые, но не все столбцы в формат с плавающей точкой. Это можно сделать и вручную, но, по счастью, в библиотеке pandas есть класс `TextParser`, который используется функцией `read_csv` и другими функциями разбора для автоматического преобразования типов:

```
from pandas.io.parsers import TextParser

def parse_options_data(table):
  rows = table.findall('.//tr')
  header = _unpack(rows[0], kind='th')
  data = [_unpack(r) for r in rows[1:]]
  return TextParser(data, names=header).get_chunk()
```

Наконец, вызываем эту функцию разбора для табличных объектов lxml и получаем результат в виде DataFrame:

```
In [920]: call_data = parse_options_data(calls)

In [921]: put_data = parse_options_data(puts)

In [922]: call_data[:10]
Out[922]:
   Strike             Symbol    Last  Chg     Bid     Ask  Vol Open Int
0     295 AAPL120818C00295000  310.40  0.0  289.80  290.80    1      169
1     300 AAPL120818C00300000  277.10  1.7  284.80  285.60    2      478
2     305 AAPL120818C00305000  300.97  0.0  279.80  280.80   10      316
3     310 AAPL120818C00310000  267.05  0.0  274.80  275.65    6      239
4     315 AAPL120818C00315000  296.54  0.0  269.80  270.80   22       88
5     320 AAPL120818C00320000  291.63  0.0  264.80  265.80   96      173
6     325 AAPL120818C00325000  261.34  0.0  259.80  260.80  N/A      108
7     330 AAPL120818C00330000  230.25  0.0  254.80  255.80  N/A       21
8     335 AAPL120818C00335000  266.03  0.0  249.80  250.65    4       46
9     340 AAPL120818C00340000  272.58  0.0  244.80  245.80    4       30
```

Разбор XML с помощью lxml.objectify

XML (расширяемый язык разметки) – еще один популярный формат представления структурированных данных, поддерживающий иерархически вложенные данные, снабженные метаданными. Текст этой книги на самом деле представляет собой набор больших XML-документов.

Выше я продемонстрировал применение библиотеки lxml и ее интерфейса `lxml.html`. А сейчас покажу альтернативный интерфейс, удобный для работы с XML-данными, – `lxml.objectify`.

Управление городского транспорта Нью-Йорка (MTA) публикует временные ряды с данными о работе автобусов и электричек (*http://www.mta.info/developers/download.html*). Мы сейчас рассмотрим данные о качестве обслуживания, хранящиеся в виде XML-файлов. Для каждой автобусной и железнодорожной компании существует свой файл (например, `Performance_MNR.xml` для компании MetroNorth Railroad), содержащий данные за один месяц в виде последовательности таких XML-записей:

```
<INDICATOR>
 <INDICATOR_SEQ>373889</INDICATOR_SEQ>
 <PARENT_SEQ></PARENT_SEQ>
 <AGENCY_NAME>Metro-North Railroad</AGENCY_NAME>
 <INDICATOR_NAME>Escalator Availability</INDICATOR_NAME>
 <DESCRIPTION>Percent of the time that escalators are operational
 systemwide. The availability rate is based on physical observations performed
 the morning of regular business days only. This is a new indicator the agency
 began reporting in 2009.</DESCRIPTION>
 <PERIOD_YEAR>2011</PERIOD_YEAR>
 <PERIOD_MONTH>12</PERIOD_MONTH>
 <CATEGORY>Service Indicators</CATEGORY>
 <FREQUENCY>M</FREQUENCY>
 <DESIRED_CHANGE>U</DESIRED_CHANGE>
 <INDICATOR_UNIT>%</INDICATOR_UNIT>
 <DECIMAL_PLACES>1</DECIMAL_PLACES>
 <YTD_TARGET>97.00</YTD_TARGET>
 <YTD_ACTUAL></YTD_ACTUAL>
 <MONTHLY_TARGET>97.00</MONTHLY_TARGET>
 <MONTHLY_ACTUAL></MONTHLY_ACTUAL>
</INDICATOR>
```

Используя `lxml.objectify`, мы разбираем файл и получаем ссылку на корневой узел XML-документа от метода `getroot`:

```
from lxml import objectify

path = 'Performance_MNR.xml'
parsed = objectify.parse(open(path))
root = parsed.getroot()
```

Свойство `root.INDICATOR` возвращает генератор, последовательно отдающий все элементы `<INDICATOR>`. Для каждой записи мы заполняем словарь имен тегов (например, `YTD_ACTUAL`) значениями данных (некоторые теги пропускаются):

```
data = []

skip_fields = ['PARENT_SEQ', 'INDICATOR_SEQ',
               'DESIRED_CHANGE', 'DECIMAL_PLACES']

for elt in root.INDICATOR:
```

```
    el_data = {}
    for child in elt.getchildren():
       if child.tag in skip_fields:
          continue
       el_data[child.tag] = child.pyval
    data.append(el_data)
```

Наконец, преобразуем этот список словарей в объект DataFrame:

```
In [927]: perf = DataFrame(data)

In [928]: perf
Out[928]:
Empty DataFrame
Columns: array([], dtype=int64)
Index: array([], dtype=int64)
```

XML-документы могут быть гораздо сложнее, чем в этом примере. В частности, в каждом элементе могут быть метаданные. Рассмотрим тег гиперссылки в формате HTML, который является частным случаем XML:

```
from StringIO import StringIO
tag = '<a href="http://www.google.com">Google</a>'

root = objectify.parse(StringIO(tag)).getroot()
```

Теперь мы можем обратиться к любому атрибуту тега (например, `href`) или к тексту ссылки:

```
In [930]: root
Out[930]: <Element a at 0x88bd4b0>

In [931]: root.get('href')
Out[931]: 'http://www.google.com'

In [932]: root.text
Out[932]: 'Google'
```

Двоичные форматы данных

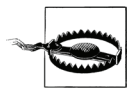
Один их самых простых способов эффективного хранения данных в двоичном формате – воспользоваться встроенным в Python методом сериализации `pickle`. Поэтому у всех объектов pandas есть метод `save`, который сохраняет данные на диске в виде pickle-файла:

```
In [933]: frame = pd.read_csv('ch06/ex1.csv')

In [934]: frame
Out[934]:

   a   b   c   d  message
```

```
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo

In [935]: frame.save('ch06/frame_pickle')
```

Прочитать данные с диска позволяет метод `pandas.load`, также упрощающий интерфейс с pickle:

```
In [936]: pd.load('ch06/frame_pickle')
Out[936]:
   a   b   c   d  message
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo
```



`pickle` рекомендуется использовать только для краткосрочного хранения. Проблема в том, что невозможно гарантировать неизменность формата: сегодня вы сериализовали объект в формате pickle, а следующая версия библиотеки не сможет его десериализовать. Я приложил все усилия к тому, чтобы в pandas такое не случалось, но, возможно, наступит момент, когда придется «поломать» формат pickle.

Формат HDF5

Существует немало инструментов, предназначенных для эффективного чтения и записи больших объемов научных данных в двоичном формате. Популярна, в частности, библиотека промышленного качества HDF5, написанная на C и имеющая интерфейсы ко многим языкам, в том числе Java, Python и MATLAB. Акроним «HDF» в ее названии означает *hierarchical data format* (иерархический формат данных). Каждый HDF5-файл содержит внутри себя структуру узлов, напоминающую файловую систему, которая позволяет хранить несколько наборов данных вместе с относящимися к ним метаданными. В отличие от более простых форматов, HDF5 поддерживает сжатие на лету с помощью различных алгоритмов сжатия, что позволяет более эффективно хранить повторяющиеся комбинации данных. Для очень больших наборов данных, которые не помещаются в память, HDF5 – отличный выбор, потому что дает возможность эффективно читать и записывать небольшие участки гораздо больших массивов.

К библиотеке HDF5 существует целых два интерфейса из Python: PyTables и h5py, в которых приняты совершенно различные подходы. h5py – прямой, хотя и высокоуровневый интерфейс к HDF5 API, тогда как PyTables абстрагирует многие детали HDF5 с целью предоставления нескольких гибких контейнеров данных, средств индексирования таблиц, средств запроса и поддержки некоторых вычислений, отсутствующих в исходной библиотеке.

В pandas имеется минимальный похожий на словарь класс `HDFStore`, в котором для сохранения объектов pandas используется интерфейс PyTables:

```
In [937]: store = pd.HDFStore('mydata.h5')

In [938]: store['obj1'] = frame

In [939]: store['obj1_col'] = frame['a']

In [940]: store
Out[940]:
<class 'pandas.io.pytables.HDFStore'>
File path: mydata.h5
obj1        DataFrame
obj1_col    Series
```

Объекты из HDF5-файла можно извлекать, как из словаря:

```
In [941]: store['obj1']
Out[941]:
   a   b   c   d  message
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo
```

Если вы собираетесь работать с очень большими объемами данных, то я рекомендую изучить PyTables и h5py и посмотреть, в какой мере они отвечают вашим потребностям. Поскольку многие задачи анализа данных ограничены, прежде всего, скоростью ввода-вывода (а не быстродействием процессора), использование средства типа HDF5 способно существенно ускорить работу приложения.

HDF5 *не является* базой данных. Лучше всего она приспособлена для работы с наборами данных, которые записываются один раз, а читаются многократно. Данные можно добавлять в файл в любой момент, но если это делают одновременно несколько клиентов, то файл можно повредить.

Чтение файлов Microsoft Excel

В pandas имеется также поддержка для чтения табличных данных в формате Excel 2003 (и более поздних версией) с помощью класса `ExcelFile`. На внутреннем уровне `ExcelFile` пользуется пакетами xlrd и openpyxl, поэтому их нужно предварительно установить. Для работы с `ExcelFile` создайте его экземпляр, передав конструктору путь к файлу с расширением `xls` или `xlsx`:

```
xls_file = pd.ExcelFile('data.xls')
```

Прочитать данные из рабочего листа в объект DataFrame позволяет метод `parse`:

```
table = xls_file.parse('Sheet1')
```

Взаимодействие с HTML и Web API

Многие сайты предоставляют открытый API для получения данных в формате JSON или каком-то другом. Получить доступ к таким API из Python можно разны-

ми способами; я рекомендую простой пакет `requests` (*http://docs.python-requests.org*). Для поиска по словам «python pandas» в Твиттере мы можем отправить такой HTTP-запрос `GET`:

```
In [944]: import requests

In [945]: url = 'http://search.twitter.com/search.json?q=python%20pandas'

In [946]: resp = requests.get(url)

In [947]: resp
Out[947]: <Response [200]>
```

У объекта Response имеет атрибут `text`, в котором хранится содержимое ответа на запрос `GET`. Многие API в веб возвращают JSON-строку, которую следует загрузить в объект Python:

```
In [948]: import json

In [949]: data = json.loads(resp.text)

In [950]: data.keys()
Out[950]:
[u'next_page',
 u'completed_in',
 u'max_id_str',
 u'since_id_str',
 u'refresh_url',
 u'results',
 u'since_id',
 u'results_per_page',
 u'query',
 u'max_id',
 u'page']
```

Поле ответа `results` содержит список твитов, каждый из которых представлен таким словарем Python:

```
{u'created_at': u'Mon, 25 Jun 2012 17:50:33 +0000',
 u'from_user': u'wesmckinn',
 u'from_user_id': 115494880,
 u'from_user_id_str': u'115494880',
 u'from_user_name': u'Wes McKinney',
 u'geo': None,
 u'id': 217313849177686018,
 u'id_str': u'217313849177686018',
 u'iso_language_code': u'pt',
 u'metadata': {u'result_type': u'recent'},
 u'source': u'<a href="http://twitter.com/">web</a>',
 u'text': u'Lunchtime pandas-fu http://t.co/SI70xZZQ #pydata',
 u'to_user': None,
 u'to_user_id': 0,
 u'to_user_id_str': u'0',
 u'to_user_name': None}
```

Далее мы можем построить список интересующих нас полей твита и передать его конструктору DataFrame:

```
In [951]: tweet_fields = ['created_at', 'from_user', 'id', 'text']

In [952]: tweets = DataFrame(data['results'], columns=tweet_fields)

In [953]: tweets
Out[953]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 15 entries, 0 to 14
Data columns:
created_at    15 non-null values
from_user     15 non-null values
id            15 non-null values
text          15 non-null values
dtypes: int64(1), object(3)
```

Теперь в каждой строке DataFrame находятся данные, извлеченные из одного твита:

```
In [121]: tweets.ix[7]
Out[121]:
created_at            Thu, 23 Jul 2012 09:54:00 +0000
from_user                                     deblike
id                                 227419585803059201
text pandas: powerful Python data analysis toolkit
Name: 7
```

Приложив толику усилий, вы сможете создать высокоуровневые интерфейсы к популярным в веб API, которые будут возвращать объекты DataFrame, легко поддающиеся анализу.

Взаимодействие с базами данных

Во многие приложения данные поступают не из файлов, потому что для хранения больших объемов данных текстовые файлы неэффективны. Широко используются реляционные базы данных на основе SQL (например, SQL Server, PostgreSQL и MySQL), а равно так называемые базы данных *NoSQL*, быстро набирающие популярность. Выбор базы данных обычно диктуется производительностью, необходимостью поддержания целостности данных и потребностями приложения в масштабируемости.

Загрузка данных из реляционной базы в DataFrame производится довольно прямолинейно, и в pandas есть несколько функций для упрощения этой процедуры. В качестве примера я возьму базу данных SQLite, целиком размещающуюся в памяти, и драйвер `sqlite3`, включенный в стандартную библиотеку Python:

```
import sqlite3

query = """
```

```
CREATE TABLE test
(a VARCHAR(20), b VARCHAR(20),
 c REAL, d INTEGER
);"""

con = sqlite3.connect(':memory:')
con.execute(query)
con.commit()
```

Затем вставлю несколько строк в таблицу:

```
data = [('Atlanta', 'Georgia', 1.25, 6),
        ('Tallahassee', 'Florida', 2.6, 3),
        ('Sacramento', 'California', 1.7, 5)]
stmt = "INSERT INTO test VALUES(?, ?, ?, ?)"

con.executemany(stmt, data)
con.commit()
```

Большинство драйверов SQL, имеющихся в Python (PyODBC, psycopg2, MySQLdb, pymssql и т. д.), при выборе данных из таблицы возвращают список кортежей:

```
In [956]: cursor = con.execute('select * from test')

In [957]: rows = cursor.fetchall()

In [958]: rows
Out[958]:
[(u'Atlanta', u'Georgia', 1.25, 6),
 (u'Tallahassee', u'Florida', 2.6, 3),
 (u'Sacramento', u'California', 1.7, 5)]
```

Этот список кортежей можно передать конструктору DataFrame, но необходимы еще имена столбцов, содержащиеся в атрибуте курсора `description`:

```
In [959]: cursor.description
Out[959]:
(('a', None, None, None, None, None, None),
 ('b', None, None, None, None, None, None),
 ('c', None, None, None, None, None, None),
 ('d', None, None, None, None, None, None))

In [960]: DataFrame(rows, columns=zip(*cursor.description)[0])
Out[960]:
             a           b      c   d
0      Atlanta     Georgia   1.25   6
1  Tallahassee     Florida   2.60   3
2   Sacramento  California   1.70   5
```

Такое переформатирование не хочется выполнять при каждом запросе к базе данных. В pandas, точнее в модуле `pandas.io.sql`, имеется функция `read_frame`, которая упрощает эту процедуру. Нужно просто передать команду `select` и объект соединения:

```
In [961]: import pandas.io.sql as sql
In [962]: sql.read_frame('select * from test', con)
Out[962]:
             a           b      c   d
0      Atlanta     Georgia   1.25   6
1  Tallahassee     Florida   2.60   3
2   Sacramento  California   1.70   5
```

Чтение и сохранение данных в MongoDB

Базы данных NoSQL весьма многообразны. Есть простые хранилища ключей и значений, напоминающие словарь, например BerkeleyDB или Tokyo Cabinet, а есть и документо-ориентированные базы, в которых похожий на словарь объект является основной единицей хранения. Я решил взять в качестве примера MongoDB (*http://mongodb.org*). Я запустил локальный экземпляр MongoDB на своей машине и подключился к ее порту по умолчанию с помощью `pymongo`, официального драйвера для MongoDB:

```
import pymongo
con = pymongo.Connection('localhost', port=27017)
```

В MongoDB документы хранятся в коллекциях в базе данных. Каждый экземпляр сервера MongoDB может обслуживать несколько баз данных, а в каждой базе может быть несколько коллекций. Допустим, я хочу сохранить данные, полученные ранее из Твиттера. Прежде всего, получаю доступ к коллекции твитов (пока пустой):

```
tweets = con.db.tweets
```

Затем запрашиваю список твитов и записываю каждый из них в коллекцию методом `tweets.save` (который сохраняет словарь Python в базе MongoDB):

```
import requests, json
url = 'http://search.twitter.com/search.json?q=python%20pandas'
data = json.loads(requests.get(url).text)

for tweet in data['results']:
  tweets.save(tweet)
```

Если затем я захочу получить все мои твиты из коллекции, то должен буду опросить ее, как показано ниже:

```
cursor = tweets.find({'from_user': 'wesmckinn'})
```

Возвращенный курсор – это итератор, который отдает каждый документ в виде словаря. Как и раньше, я могу преобразовать этот документ в DataFrame, возможно, ограничившись некоторым подмножеством полей данных:

```
tweet_fields = ['created_at', 'from_user', 'id', 'text']
result = DataFrame(list(cursor), columns=tweet_fields)
```

ГЛАВА 7. Переформатирование данных: очистка, преобразование, слияние, изменение формы

Значительная часть времени программиста, занимающегося анализом и моделированием данных, уходит на подготовку данных: загрузку, очистку, преобразование и реорганизацию. Иногда способ хранения данных в файлах или в базе не согласуется с алгоритмом обработки. Многие предпочитают писать преобразования данных из одной формы в другую на каком-нибудь универсальном языке программирования типа Python, Perl, R или Java либо с помощью имеющихся в UNIX средств обработки текста типа sed или awk. По счастью, pandas дополняет стандартную библиотеку Python высокоуровневыми, гибкими и производительными базовыми преобразованиями и алгоритмами, которые позволяют переформатировать данные без особых проблем.

Если вы наткнетесь на манипуляцию, которой нет ни в этой книге, ни вообще в библиотеке pandas, не стесняйтесь внести предложение в списке рассылки или на сайте GitHub. Вообще, многое в pandas – в части как проектирования, так и реализации – обусловлено потребностями реальных приложений.

Комбинирование и слияние наборов данных

Данные, хранящиеся в объектах pandas, можно комбинировать различными готовыми способами:

- Метод `pandas.merge` соединяет строки объектов DataFrame по одному или нескольким ключам. Эта операция хорошо знакома пользователям реляционных баз данных.
- Метод `pandas.concat` «склеивает» объекты, располагая их в стопку вдоль оси.
- Метод экземпляра `combine_first` позволяет сращивать перекрывающиеся данные, чтобы заполнить отсутствующие в одном объекте данные значениями из другого объекта.

Я рассмотрю эти способы на многочисленных примерах. Мы будем неоднократно пользоваться ими в последующих главах.

Слияние объектов DataFrame как в базах данных

Операция *слияния* или *соединения* комбинирует наборы данных, соединяя строки по одному или нескольким *ключам*. Эта операция является одной из основных в базах данных. Функция `merge` в pandas – портал ко всем алгоритмам такого рода.

Начнем с простого примера:

```
In [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})

In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [17]: df1                        In [18]: df2
Out[17]:                            Out[18]:
   data1  key                          data2  key
0      0    b                       0      0    a
1      1    b                       1      1    b
2      2    a                       2      2    d
3      3    c
4      4    a
5      5    a
6      6    b
```

Это пример слияния типа *многие-к-одному*; в объекте `df1` есть несколько строк с метками `a` и `b`, а в `df2` – только одна строка для каждого значения в столбце `key`. Вызов `merge` для таких объектов дает:

```
In [19]: pd.merge(df1, df2)
Out[19]:
  data1  key  data2
0     2    a      0
1     4    a      0
2     5    a      0
3     0    b      1
4     1    b      1
5     6    b      1
```

Обратите внимание, что я не указал, по какому столбцу производить соединение. В таком случае `merge` использует в качестве ключей столбцы с одинаковыми именами. Однако рекомендуется все же указывать столбцы явно:

```
In [20]: pd.merge(df1, df2, on='key')
Out[20]:
  data1  key  data2
0     2    a      0
1     4    a      0
2     5    a      0
3     0    b      1
```

```
4     1    b      1
5     6    b      1
```

Если имена столбцов в объектах различаются, то можно задать их порознь:

```
In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
    ....:                 'data1': range(7)})

In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'],
    ....:                 'data2': range(3)})

In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[23]:
  data1 lkey  data2 rkey
0     2    a      0    a
1     4    a      0    a
2     5    a      0    a
3     0    b      1    b
4     1    b      1    b
5     6    b      1    b
```

Вероятно, вы обратили внимание, что значения `'c'` и `'d'` и ассоциированные с ними данные отсутствуют в результирующем объекте. По умолчанию функция `merge` производит внутреннее соединение (`'inner'`); в результирующий объект попадают только ключи, присутствующие в обоих объектах-аргументах. Альтернативы: `'left'`, `'right'` и `'outer'`. В случае внешнего соединения (`'outer'`) берется объединение ключей, т. е. получается то же самое, что при совместном применении левого и правого соединения:

```
In [24]: pd.merge(df1, df2, how='outer')
Out[24]:
   data1  key  data2
0      2    a      0
1      4    a      0
2      5    a      0
3      0    b      1
4      1    b      1
5      6    b      1
6      3    c    NaN
7    NaN    d      2
```

Для слияния типа *многие-ко-многим* поведение корректно определено, хотя на первый взгляд неочевидно. Вот пример:

```
In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
    ....:                 'data1': range(6)})
In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
    ....:                 'data2': range(5)})

In [27]: df1                    In [28]: df2
Out[27]:                        Out[28]:
   data1  key                      data2  key
0      0    b                   0      0    a
```

```
1      1    b                1      1    b
2      2    a                2      2    a
3      3    c                3      3    b
4      4    a                4      4    d
5      5    b

In [29]: pd.merge(df1, df2, on='key', how='left')
Out[29]:
   data1  key  data2
0      2    a      0
1      2    a      2
2      4    a      0
3      4    a      2
4      0    b      1
5      0    b      3
6      1    b      1
7      1    b      3
8      5    b      1
9      5    b      3
10     3    c    NaN
```

Соединение многие-ко-многим порождает декартово произведение строк. Поскольку в левом объекте DataFrame было три строки с ключом `'b'`, а в правом – две, то в результирующем объекте таких строк получилось шесть. Метод соединения оказывает влияние только на множество различных ключей в результате:

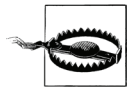
```
In [30]: pd.merge(df1, df2, how='inner')
Out[30]:
   data1  key  data2
0      2    a      0
1      2    a      2
2      4    a      0
3      4    a      2
4      0    b      1
5      0    b      3
6      1    b      1
7      1    b      3
8      5    b      1
9      5    b      3
```

Для слияния по нескольким ключам следует передать список имен столбцов:

```
In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                   'key2': ['one', 'two', 'one'],
   ....:                   'lval': [1, 2, 3]})

In [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                    'key2': ['one', 'one', 'one', 'two'],
   ....:                    'rval': [4, 5, 6, 7]})

In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[33]:
   key1  key2  lval  rval
0   bar   one     3     6
```

```
1   bar   two   NaN     7
2   foo   one     1     4
3   foo   one     1     5
4   foo   two     2   NaN
```

Чтобы определить, какие комбинации ключей появятся в результате при данном выборе метода слияния, полезно представить несколько ключей как массив кортежей, используемый в качестве единственного ключа соединения (хотя на самом деле операция реализована не так).



При соединении по столбцам индексы над переданными объектами DataFrame отбрасываются.

Последний момент, касающийся операций слияния, – обработка одинаковых имен столбцов. Хотя эту проблему можно решить вручную (см. раздел о переименовании меток на осях ниже), у функции `merge` имеется параметр `suffixes`, позволяющий задать строки, которые должны дописываться в конец одинаковых имен в левом и правом объектах DataFrame:

```
In [34]: pd.merge(left, right, on='key1')
Out[34]:
   key1  key2_x  lval  key2_y  rval
0   bar     one     3     one     6
1   bar     one     3     two     7
2   foo     one     1     one     4
3   foo     one     1     one     5
4   foo     two     2     one     4
5   foo     two     2     one     5

In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[35]:
   key1  key2_left  lval  key2_right  rval
0   bar        one     3         one     6
1   bar        one     3         two     7
2   foo        one     1         one     4
3   foo        one     1         one     5
4   foo        two     2         one     4
5   foo        two     2         one     5
```

В табл. 7.1 приведена справка по аргументам функции `merge`. Соединение по индексу – тема следующего раздела.

Таблица 7.1. Аргументы функции `merge`

Аргумент	Описание
`left`	Объект DataFrame в левой части операции слияния
`right`	Объект DataFrame в правой части операции слияния
`how`	Допустимые значения: `'inner'`, `'outer'`, `'left'`, `'right'`.

Аргумент	Описание
on	Имена столбцов, по которым производится соединение. Должны присутствовать в обоих объектах DataFrame. Если не заданы и не указаны никакие другие ключи соединения, то используются имена столбцов, общих для обоих объектов
left_on	Столбцы левого DataFrame, используемые как ключи соединения
right_on	Столбцы правого DataFrame, используемые как ключи соединения
left_index	Использовать индекс строк левого DataFrame в качестве его ключа соединения (или нескольких ключей в случае мультииндекса)
right_index	То же, что left_index, но для правого DataFrame
sort	Сортировать слитые данные лексикографически по ключам соединения; по умолчанию True. Иногда при работе с большими наборами данных лучше отключить
suffixes	Кортеж строк, которые дописываются в конец совпадающих имен столбцов; по умолчанию ('_x', '_y'). Например, если в обоих объектах DataFrame встречается столбец 'data', то в результирующем объекте появятся столбцы 'data_x' и 'data_y'
copy	Если равен False, то в некоторых особых случаях разрешается не копировать данные в результирующую структуру. По умолчанию данные копируются всегда

Слияние по индексу

Иногда ключ (или ключи) слияния находится в индексе объекта DataFrame. В таком случае можно задать параметр left_index=True или right_index=True (или то и другое), чтобы указать, что в качестве ключа слияния следует использовать индекс:

```
In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                    'value': range(6)})

In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [38]: left1
Out[38]:
   key  value
0    a      0
1    b      1
2    a      2
3    a      3
4    b      4
5    c      5

In [39]: right1
Out[39]:
   group_val
a        3.5
b        7.0

In [40]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[40]:
   key  value group_val
0    a      0       3.5
2    a      2       3.5
```

```
3    a      3       3.5
1    b      1       7.0
4    b      4       7.0
```

По умолчанию слияние производится по пересекающимся ключам, но можно вместо пересечения выполнить объединение, указав внешнее соединение:

```
In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[41]:
   key  value group_val
0    a      0       3.5
2    a      2       3.5
3    a      3       3.5
1    b      1       7.0
4    b      4       7.0
5    c      5       NaN
```

В случае иерархически индексированных данных ситуация немного усложняется:

```
In [42]: lefth = DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
   ....:                    'key2': [2000, 2001, 2002, 2001, 2002],
   ....:                    'data': np.arange(5.)})

In [43]: righth = DataFrame(np.arange(12).reshape((6, 2)),
   ....:                 index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
   ....:                        [2001, 2000, 2000, 2000, 2001, 2002]],
   ....:                 columns=['event1', 'event2'])

In [44]: lefth                    In [45]: righth
Out[44]:                          Out[45]:
   data    key1  key2                           event1  event2
0     0    Ohio  2000          Nevada 2001           0       1
1     1    Ohio  2001                 2000           2       3
2     2    Ohio  2002          Ohio   2000           4       5
3     3  Nevada  2001                 2000           6       7
4     4  Nevada  2002                 2001           8       9
                                      2002          10      11
```

В данном случае необходимо перечислить столбцы, по которым производится слияние, в виде списка (обращайте внимание на обработку повторяющихся значений в индексе):

```
In [46]: pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True)
Out[46]:
   data    key1  key2  event1  event2
3     3  Nevada  2001       0       1
0     0    Ohio  2000       4       5
0     0    Ohio  2000       6       7
1     1    Ohio  2001       8       9
2     2    Ohio  2002      10      11

In [47]: pd.merge(lefth, righth, left_on=['key1', 'key2'],
```

```
   ....:           right_index=True, how='outer')
Out[47]:
   data    key1  key2  event1  event2
4   NaN  Nevada  2000       2       3
3     3  Nevada  2001       0       1
4     4  Nevada  2002     NaN     NaN
0     0    Ohio  2000       4       5
0     0    Ohio  2000       6       7
1     1    Ohio  2001       8       9
2     2    Ohio  2002      10      11
```

Употребление индексов в обеих частях слияния тоже не проблема:

```
In [48]: left2 = DataFrame([[1., 2.], [3., 4.], [5., 6.]], index=['a', 'c', 'e'],
   ....:                   columns=['Ohio', 'Nevada'])

In [49]: right2 = DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
   ....:                    index=['b', 'c', 'd', 'e'], columns=['Missouri', 'Alabama'])

In [50]: left2
Out[50]:
   Ohio  Nevada
a     1       2
c     3       4
e     5       6

In [51]: right2
Out[51]:
   Missouri  Alabama
b         7        8
c         9       10
d        11       12
e        13       14

In [52]: pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
Out[52]:
   Ohio  Nevada  Missouri  Alabama
a     1       2       NaN      NaN
b   NaN     NaN         7        8
c     3       4         9       10
d   NaN     NaN        11       12
e     5       6        13       14
```

В классе DataFrame есть и более удобный метод экземпляра `join` для слияния по индексу. Его также можно использовать для комбинирования нескольких объектов DataFrame, обладающих одинаковыми или похожими индексами, но непересекающимися столбцами. В предыдущем примере можно было бы написать:

```
In [53]: left2.join(right2, how='outer')
Out[53]:
   Ohio  Nevada  Missouri  Alabama
a     1       2       NaN      NaN
b   NaN     NaN         7        8
c     3       4         9       10
d   NaN     NaN        11       12
e     5       6        13       14
```

Отчасти из-за необходимости поддерживать совместимость (с очень старыми версиями pandas), метод `join` объекта DataFrame выполняет левое внешнее соединение. Он также поддерживает соединение с индексом переданного DataFrame по одному из столбцов вызывающего:

```
In [54]: left1.join(right1, on='key')
Out[54]:
   key  value group_val
0    a      0       3.5
1    b      1       7.0
2    a      2       3.5
3    a      3       3.5
4    b      4       7.0
5    c      5       NaN
```

Наконец, в случае простых операций слияния индекса с индексом можно передать список объектов DataFrame методу `join` в качестве альтернативы использованию более общей функции `concat`, которая описана ниже:

```
In [55]: another = DataFrame([[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
   ....:              index=['a', 'c', 'e', 'f'], columns=['New York', 'Oregon'])

In [56]: left2.join([right2, another])
Out[56]:
   Ohio Nevada Missouri  Alabama  New York  Oregon
a     1      2      NaN      NaN         7       8
c     3      4        9       10         9      10
e     5      6       13       14        11      12

In [57]: left2.join([right2, another], how='outer')
Out[57]:
   Ohio Nevada Missouri  Alabama  New York  Oregon
a     1      2      NaN      NaN         7       8
b   NaN    NaN        7        8       NaN     NaN
c     3      4        9       10         9      10
d   NaN    NaN       11       12       NaN     NaN
e     5      6       13       14        11      12
f   NaN    NaN      NaN      NaN        16      17
```

Конкатенация вдоль оси

Еще одну операцию комбинирования данных разные авторы называют по-разному: конкатенация, связывание или укладка. В библиотеке NumPy имеется функция `concatenate` для выполнения этой операции над массивами:

```
In [58]: arr = np.arange(12).reshape((3, 4))

In [59]: arr
Out[59]:
array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])

In [60]: np.concatenate([arr, arr], axis=1)
Out[60]:
array([[ 0, 1, 2, 3, 0, 1, 2, 3],
       [ 4, 5, 6, 7, 4, 5, 6, 7],
       [ 8, 9, 10, 11, 8, 9, 10, 11]])
```

В контексте объектов pandas, Series и DataFrame наличие помеченных осей позволяет обобщить конкатенацию массивов. В частности, нужно решить следующие вопросы:

- Если объекты по-разному проиндексированы по другим осям, то как поступать с коллекцией осей: объединять или пересекать?
- Нужно ли иметь возможность идентифицировать группы в результирующем объекте?
- Имеет ли вообще какое-то значение ось конкатенации?

Функция `concat` в pandas дает согласованные ответы на эти вопросы. Я покажу, как она работает, на примерах. Допустим, имеются три объекта Series с непересекающимися индексами:

```
In [61]: s1 = Series([0, 1], index=['a', 'b'])

In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])

In [63]: s3 = Series([5, 6], index=['f', 'g'])
```

Если передать их функции `concat` списком, то она «склеит» данные и индексы:

```
In [64]: pd.concat([s1, s2, s3])
Out[64]:
a  0
b  1
c  2
d  3
e  4
f  5
g  6
```

По умолчанию `concat` работает вдоль оси `axis=0`, порождая новый объект Series. Но если передать параметр `axis=1`, то результатом будет DataFrame (в нем `axis=1` – ось столбцов):

```
In [65]: pd.concat([s1, s2, s3], axis=1)
Out[65]:
     0    1    2
a    0  NaN  NaN
b    1  NaN  NaN
c  NaN    2  NaN
d  NaN    3  NaN
e  NaN    4  NaN
f  NaN  NaN    5
g  NaN  NaN    6
```

В данном случае на другой оси нет перекрытия, и она, как видно, является отсортированным объединением (внешним соединением) индексов. Но можно образовать и пересечение индексов, если передать параметр `join='inner'`:

```
In [66]: s4 = pd.concat([s1 * 5, s3])

In [67]: pd.concat([s1, s4], axis=1)
Out[67]:
     0  1
a    0  0
b    1  5
f  NaN  5
g  NaN  6

In [68]: pd.concat([s1, s4], axis=1, join='inner')
Out[68]:
     0  1
a    0  0
b    1  5
```

Можно даже задать, какие метки будут использоваться на других осях – с помощью параметра `join_axes`:

```
In [69]: pd.concat([s1, s4], axis=1, join_axes=[['a', 'c', 'b', 'e']])
Out[69]:
     0    1
a    0    0
c  NaN  NaN
b    1    5
e  NaN  NaN
```

Проблема возникает из-за того, что в результирующем объекте не видно, конкатенацией каких объектов он получен. Допустим, что вместо этого требуется построить иерархический индекс на оси конкатенации. Для этого используется аргумент `keys`:

```
In [70]: result = pd.concat([s1, s1, s3], keys=['one', 'two', 'three'])

In [71]: result
Out[71]:
one    a  0
       b  1
two    a  0
       b  1
three  f  5
       g  6

# О функции unstack будет рассказано гораздо подробнее ниже
In [72]: result.unstack()
Out[72]:
         a    b    f    g
one      0    1  NaN  NaN
two      0    1  NaN  NaN
three  NaN  NaN    5    6
```

При комбинировании Series вдоль оси `axis=1` элементы списка `keys` становятся заголовками столбцов объекта DataFrame:

```
In [73]: pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
Out[73]:
   one  two  three
a    0  NaN    NaN
b    1  NaN    NaN
```

```
c  NaN    2    NaN
d  NaN    3    NaN
e  NaN    4    NaN
f  NaN  NaN      5
g  NaN  NaN      6
```

Эта логика обобщается и на объекты DataFrame:

```
In [74]: df1 = DataFrame(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'],
   ....:                 columns=['one', 'two'])

In [75]: df2 = DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'],
   ....:                 columns=['three', 'four'])

In [76]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
Out[76]:
   level1       level2
      one  two   three  four
a       0    1       5     6
b       2    3     NaN   NaN
c       4    5       7     8
```

Если передать не список, а словарь объектов, то роль аргумента `keys` будут играть ключи словаря:

```
In [77]: pd.concat({'level1': df1, 'level2': df2}, axis=1)
Out[77]:
   level1       level2
      one  two   three  four
a       0    1       5     6
b       2    3     NaN   NaN
c       4    5       7     8
```

Есть еще два аргумента, управляющие созданием иерархического индекса (см. табл. 7.2):

```
In [78]: pd.concat([df1, df2], axis=1, keys=['level1', 'level2'],
   ....:           names=['upper', 'lower'])
Out[78]:
upper   level1       level2
lower      one  two   three  four
a            0    1       5     6
b            2    3     NaN   NaN
c            4    5       7     8
```

Последнее замечание касается объектов DataFrame, в которых индекс строк не имеет смысла в контексте анализа:

```
In [79]: df1 = DataFrame(np.random.randn(3, 4), columns=['a', 'b', 'c', 'd'])

In [80]: df2 = DataFrame(np.random.randn(2, 3), columns=['b', 'd', 'a'])

In [81]: df1                                    In [82]: df2
Out[81]:                                        Out[82]:
```

```
          a         b         c         d                b         d         a
0 -0.204708 0.478943 -0.519439 -0.555730      0  0.274992  0.228913  1.352917
1  1.965781 1.393406  0.092908  0.281746      1  0.886429 -2.001637 -0.371843
2  0.769023 1.246435  1.007189 -1.296221
```

В таком случае можно передать параметр `ignore_index=True`:

```
In [83]: pd.concat([df1, df2], ignore_index=True)
Out[83]:
          a         b         c          d
0  -0.204708 0.478943 -0.519439  -0.555730
1   1.965781 1.393406  0.092908   0.281746
2   0.769023 1.246435  1.007189  -1.296221
3   1.352917 0.274992       NaN   0.228913
4  -0.371843 0.886429       NaN  -2.001637
```

Таблица 7.2. Аргументы функции `concat`

Аргумент	Описание
`objs`	Список или словарь конкатенируемых объектов pandas. Единственный обязательный аргумент.
`axis`	Ось, вдоль которой производится конкатенация, по умолчанию 0
`join`	Допустимые значения: `'inner'`, `'outer'`, по умолчанию `'outer'`; следует ли пересекать (`inner`) или объединять (`outer`) индексы вдоль других осей.
`join_axes`	Какие конкретно индексы использовать для других *n*–1 осей вместо выполнения пересечения или объединения
`keys`	Значения, которые ассоциируются с конкатенируемыми объектами и образуют иерархический индекс вдоль оси конкатенации. Может быть список или массив произвольных значений, а также массив кортежей или список массивов (если в параметре `levels` передаются массивы для нескольких уровней)
`levels`	Конкретные индексы, которые используются на одном или нескольких уровнях иерархического индекса, если задан параметр `keys`
`names`	Имена создаваемых уровней иерархического индекса, если заданы параметры `keys` и (или) `levels`
`verify_integrity`	Проверить новую ось в конкатенированном объекте на наличие дубликатов и, если они имеются, возбудить исключение. По умолчанию `False` – дубликаты разрешены
`ignore_index`	Не сохранять индексы вдоль оси конкатенации, а вместо этого создать новый индекс `range(total_length)`

Комбинирование перекрывающихся данных

Есть еще одна ситуация, которую нельзя выразить как слияние или конкатенацию. Речь идет о двух наборах данных, индексы которых полностью или частично пересекаются. В качестве пояснительного примера рассмотрим функцию NumPy `where`, которая выражает векторный аналог if-else:

```
In [84]: a = Series([np.nan, 2.5, np.nan, 3.5, 4.5, np.nan],
    ....: index=['f', 'e', 'd', 'c', 'b', 'a'])

In [85]: b = Series(np.arange(len(a), dtype=np.float64),
   ....: index=['f', 'e', 'd', 'c', 'b', 'a'])

In [86]: b[-1] = np.nan

In [87]: a      In [88]: b      In [89]: np.where(pd.isnull(a), b, a)
Out[87]:        Out[88]:        Out[89]:
f    NaN        f     0         f    0.0
e    2.5        e     1         e    2.5
d    NaN        d     2         d    2.0
c    3.5        c     3         c    3.5
b    4.5        b     4         b    4.5
a    NaN        a   NaN         a    NaN
```

У объекта Series имеется метод `combine_first`, который выполняет эквивалент этой операции плюс выравнивание данных:

```
In [90]: b[:-2].combine_first(a[2:])
Out[90]:
a    NaN
b    4.5
c    3.0
d    2.0
e    1.0
f    0.0
```

В случае DataFrame метод `combine_first` делает то же самое для каждого столбца, так что можно считать, что он «подставляет» вместо данных, отсутствующих в вызывающем объекте, данные из объекта, переданного в аргументе:

```
In [91]: df1 = DataFrame({'a': [1., np.nan, 5., np.nan],
   ....:                  'b': [np.nan, 2., np.nan, 6.],
   ....:                  'c': range(2, 18, 4)})

In [92]: df2 = DataFrame({'a': [5., 4., np.nan, 3., 7.],
   ....: 'b': [np.nan, 3., 4., 6., 8.]})

In [93]: df1.combine_first(df2)
Out[93]:
   a    b    c
0  1  NaN    2
1  4    2    6
2  5    4   10
3  3    6   14
4  7    8  NaN
```

Изменение формы и поворот

Существует ряд фундаментальных операций реорганизации табличных данных. Иногда их называют *изменением формы* (reshape), а иногда – *поворотом* (pivot).

Изменение формы с помощью иерархического индексирования

Иерархическое индексирование дает естественный способ реорганизовать данные в DataFrame. Есть два основных действия:

- `stack`: это «поворот», который переносит данные из столбцов в строки;
- `unstack`: обратный поворот, который переносит данные из строк в столбцы.

Проиллюстрирую эти операции на примерах. Рассмотрим небольшой DataFrame, в котором индексы строк и столбцов – массивы строк.

```
In [94]: data = DataFrame(np.arange(6).reshape((2, 3)),
   ....:                  index=pd.Index(['Ohio', 'Colorado'], name='state'),
   ....:                  columns=pd.Index(['one', 'two', 'three'], name='number'))

In [95]: data
Out[95]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

Метод `stack` поворачивает таблицу, так что столбцы оказываются строками, и в результате получается объект Series:

```
In [96]: result = data.stack()

In [97]: result
Out[97]:
state     number
Ohio      one     0
          two     1
          three   2
Colorado  one     3
          two     4
          three   5
```

Имея иерархически проиндексированный объект Series, мы можем восстановить DataFrame методом `unstack`:

```
In [98]: result.unstack()
Out[98]:
number    one  two  three
state
Ohio        0    1      2
Colorado    3    4      5
```

По умолчанию поворачивается самый внутренний уровень (как и в случае `stack`). Но можно повернуть и любой другой, если указать номер или имя уровня:

```
In [99]: result.unstack(0)
Out[99]:
```

```
In [100]: result.unstack('state')
Out[100]:
```

```
state   Ohio  Colorado
number
one        0         3
two        1         4
three      2         5
```

```
state    Ohio Colorado
number
one         0        3
two         1        4
three       2        5
```

При обратном повороте могут появиться отсутствующие данные, если не каждое значение на указанном уровне присутствует в каждой подгруппе:

```
In [101]: s1 = Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])

In [102]: s2 = Series([4, 5, 6], index=['c', 'd', 'e'])

In [103]: data2 = pd.concat([s1, s2], keys=['one', 'two'])

In [104]: data2.unstack()
Out[104]:
       a    b  c  d    e
one    0    1  2  3  NaN
two  NaN  NaN  4  5    6
```

При выполнении поворота отсутствующие данные по умолчанию отфильтровываются, поэтому операция обратима:

```
In [105]: data2.unstack().stack()
Out[105]:
one  a 0
     b 1
     c 2
     d 3
two  c 4
     d 5
     e 6
```

```
In [106]: data2.unstack().stack(dropna=False)
Out[106]:
one a   0
    b   1
    c   2
    d   3
    e NaN
two a NaN
    b NaN
    c   4
    d   5
    e   6
```

В случае обратного поворота DataFrame поворачиваемый уровень становится самым нижним уровнем результирующего объекта:

```
In [107]: df = DataFrame({'left': result, 'right': result + 5},
   .....:                 columns=pd.Index(['left', 'right'], name='side'))

In [108]: df
Out[108]:
side              left  right
state    number
Ohio     one         0      5
         two         1      6
         three       2      7
Colorado one         3      8
         two         4      9
         three       5     10

In [109]: df.unstack('state')
```

```
In [110]: df.unstack('state').stack('side')
```

```
Out[109]:
side    left           right
state   Ohio Colorado  Ohio Colorado
number
one        0        3     5        8
two        1        4     6        9
three      2        5     7       10
```

```
Out[110]:
state        Ohio  Colorado
number  side
one     left    0         3
        right   5         8
two     left    1         4
        right   6         9
three   left    2         5
        right   7        10
```

Поворот из «длинного» в «широкий» формат

Стандартный способ хранения нескольких временных рядов в базах данных и в CSV-файлах – так называемый *длинный* формат (*в столбик*):

```
In [116]: ldata[:10]
Out[116]:
                  date     item     value
0 1959-03-31 00:00:00  realgdp  2710.349
1 1959-03-31 00:00:00     infl     0.000
2 1959-03-31 00:00:00    unemp     5.800
3 1959-06-30 00:00:00  realgdp  2778.801
4 1959-06-30 00:00:00     infl     2.340
5 1959-06-30 00:00:00    unemp     5.100
6 1959-09-30 00:00:00  realgdp  2775.488
7 1959-09-30 00:00:00     infl     2.740
8 1959-09-30 00:00:00    unemp     5.300
9 1959-12-31 00:00:00  realgdp  2785.204
```

Так данные часто хранятся в реляционных базах данных типа MySQL, поскольку при наличии фиксированной схемы (совокупность имен и типов данных столбцов) количество различных значений в столбце `item` может увеличиваться или уменьшаться при добавлении или удалении данных. В примере выше пара столбцов `date` и `item` обычно выступает в роли первичного ключа (в терминологии реляционных баз данных), благодаря которому обеспечивается целостность данных и упрощаются многие операции соединения и запросы. Но есть и минус: с данными в длинном формате трудно работать; иногда предпочтительнее иметь объект DataFrame, содержащий по одному столбцу на каждое уникальное значение `item` и проиндексированный временными метками в столбце `date`. Метод `pivot` объекта DataFrame именно такое преобразование и выполняет:

```
In [117]: pivoted = ldata.pivot('date', 'item', 'value')

In [118]: pivoted.head()
Out[118]:

item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
```

```
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

Первые два аргумента – столбцы, которые будут выступать в роли индексов строк и столбцов, а последний необязательный аргумент – столбец, в котором находятся данные, вставляемые в DataFrame. Допустим, что имеется два столбца значений, форму которых требуется изменить одновременно:

```
In [119]: ldata['value2'] = np.random.randn(len(ldata))

In [120]: ldata[:10]
Out[120]:
                 date     item     value     value2
0 1959-03-31 00:00:00  realgdp  2710.349   1.669025
1 1959-03-31 00:00:00     infl     0.000  -0.438570
2 1959-03-31 00:00:00    unemp     5.800  -0.539741
3 1959-06-30 00:00:00  realgdp  2778.801   0.476985
4 1959-06-30 00:00:00     infl     2.340   3.248944
5 1959-06-30 00:00:00    unemp     5.100  -1.021228
6 1959-09-30 00:00:00  realgdp  2775.488  -0.577087
7 1959-09-30 00:00:00     infl     2.740   0.124121
8 1959-09-30 00:00:00    unemp     5.300   0.302614
9 1959-12-31 00:00:00  realgdp  2785.204   0.523772
```

Опустив последний аргумент, мы получим DataFrame с иерархическими столбцами:

```
In [121]: pivoted = ldata.pivot('date', 'item')

In [122]: pivoted[:5]
Out[122]:
           value                          value2
item        infl    realgdp  unemp        infl     realgdp       unemp
date
1959-03-31  0.00  2710.349    5.8  -0.438570    1.669025  -0.539741
1959-06-30  2.34  2778.801    5.1   3.248944    0.476985  -1.021228
1959-09-30  2.74  2775.488    5.3   0.124121   -0.577087   0.302614
1959-12-31  0.27  2785.204    5.6   0.000940    0.523772   1.343810
1960-03-31  2.31  2847.699    5.2  -0.831154   -0.713544  -2.370232

In [123]: pivoted['value'][:5]
Out[123]:
item        infl    realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

Отметим, что метод `pivot` – это не более чем сокращенный способ создания иерархического индекса с помощью `set_index` и последующего изменения формы с помощью `unstack`:

```
In [124]: unstacked = ldata.set_index(['date', 'item']).unstack('item')

In [125]: unstacked[:7]
Out[125]:
value value2
item infl realgdp unemp infl realgdp unemp
date
```

	value			value2		
item	infl	realgdp	unemp	infl	realgdp	unemp
date						
1959-03-31	0.00	2710.349	5.8	-0.438570	1.669025	-0.539741
1959-06-30	2.34	2778.801	5.1	3.248944	0.476985	-1.021228
1959-09-30	2.74	2775.488	5.3	0.124121	-0.577087	0.302614
1959-12-31	0.27	2785.204	5.6	0.000940	0.523772	1.343810
1960-03-31	2.31	2847.699	5.2	-0.831154	-0.713544	-2.370232

Преобразование данных

До сих пор мы в этой главе занимались реорганизацией данных. Фильтрация, очистка и прочие преобразования составляют еще один, не менее важный, класс операций.

Устранение дубликатов

Строки-дубликаты могут появиться в объекте DataFrame по разным причинам. Приведем пример:

```
In [126]: data = DataFrame({'k1': ['one'] * 3 + ['two'] * 4,
   .....:                   'k2': [1, 1, 2, 3, 3, 4, 4]})
In [127]: data
Out[127]:
    k1 k2
0  one  1
1  one  1
2  one  2
3  two  3
4  two  3
5  two  4
6  two  4
```

Метод `duplicated` возвращает булев объект Series, который для каждой строки показывает, есть в ней дубликаты или нет:

```
In [128]: data.duplicated()
Out[128]:
0   False
1    True
2   False
3   False
4    True
5   False
6    True
```

А метод `drop_duplicates` возвращает DataFrame, для которого массив `duplicated` будет содержать только значения `True`:

```
In [129]: data.drop_duplicates()
Out[129]:
    k1 k2
0  one  1
2  one  2
3  two  3
5  two  4
```

По умолчанию оба метода принимают во внимание все столбцы, но можно указать произвольное подмножество столбцов, которые необходимо исследовать на наличие дубликатов. Допустим, есть еще один столбец значений, и мы хотим отфильтровать строки, которые содержат повторяющиеся значения в столбце `'k1'`:

```
In [130]: data['v1'] = range(7)
In [131]: data.drop_duplicates(['k1'])
Out[131]:
    k1  k2  v1
0  one   1   0
3  two   3   3
```

По умолчанию методы `duplicated` и `drop_duplicates` оставляют первую встретившуюся строку с данной комбинацией значений. Но если задать параметр `take_last=True`, то будет оставлена последняя строка:

```
In [132]: data.drop_duplicates(['k1', 'k2'], take_last=True)
Out[132]:
    k1  k2  v1
1  one   1   1
2  one   2   2
4  two   3   4
6  two   4   6
```

Преобразование данных с помощью функции или отображения

Часто бывает необходимо произвести преобразование набора данных, исходя из значений в некотором массиве, объекте Series или столбце объекта DataFrame. Рассмотрим гипотетические данные о сортах мяса:

```
In [133]: data = DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami',
   .....:                   'corned beef', 'Bacon', 'pastrami', 'honey ham',
   .....:                   'nova lox'],
   .....:                   'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

In [134]: data
Out[134]:
          food  ounces
0        bacon     4.0
1  pulled pork     3.0
2        bacon    12.0
```

```
3     Pastrami     6.0
4  corned beef     7.5
5        Bacon     8.0
6     pastrami     3.0
7    honey ham     5.0
8     nova lox     6.0
```

Допустим, что требуется добавить столбец, в котором указано соответствующее сорту мяса животное. Создадим отображение сортов мяса на виды животных:

```
meat_to_animal = {
  'bacon'      : 'pig',
  'pulled pork': 'pig',
  'pastrami':    'cow',
  'corned beef': 'cow',
  'honey ham':   'pig',
  'nova lox':    'salmon'
}
```

Метод `map` объекта Series принимает функцию или похожий на словарь объект, содержащий отображений, но в данном случае возникает мелкая проблема: у нас названия некоторых сортов мяса начинаются с заглавной буквы, а остальные – со строчной. Поэтому нужно привести все строки к нижнему регистру:

```
In [136]: data['animal'] = data['food'].map(str.lower).map(meat_to_animal)

In [137]: data
Out[137]:
          food ounces  animal
0        bacon   4.0      pig
1  pulled pork   3.0      pig
2        bacon  12.0      pig
3     Pastrami   6.0      cow
4  corned beef   7.5      cow
5        Bacon   8.0      pig
6     pastrami   3.0      cow
7    honey ham   5.0      pig
8     nova lox   6.0   salmon
```

Можно было бы также передать функцию, выполняющую всю эту работу:

```
In [138]: data['food'].map(lambda x: meat_to_animal[x.lower()])
Out[138]:
0      pig
1      pig
2      pig
3      cow
4      cow
5      pig
6      cow
7      pig
8   salmon
Name: food
```

Метод map – удобное средство выполнения поэлементных преобразований и других операций очистки.

Замена значений

Восполнение отсутствующих данных методом fillna можно рассматривать как частный случай более общей замены значений. Если метод map, как мы только что видели, позволяет модифицировать подмножество значений, хранящихся в объекте, то метод replace предлагает для этого более простой и гибкий интерфейс. Рассмотрим такой объект Series:

```
In [139]: data = Series([1., -999., 2., -999., -1000., 3.])

In [140]: data
Out[140]:
0        1
1     -999
2        2
3     -999
4    -1000
5        3
```

Значение -999 могло бы быть маркером отсутствующих данных. Чтобы заменить все такие значения теми, которые понимает pandas, воспользуемся методом replace, порождающим новый объект Series:

```
In [141]: data.replace(-999, np.nan)
Out[141]:
0      1
1    NaN
2      2
3    NaN
4  -1000
5      3
```

Чтобы заменить сразу несколько значений, нужно передать их список и заменяющее значение:

```
In [142]: data.replace([-999, -1000], np.nan)
Out[142]:
0      1
1    NaN
2      2
3    NaN
4    NaN
5      3
```

Если для каждого заменяемого значения нужно свое заменяющее, передаем список замен:

```
In [143]: data.replace([-999, -1000], [np.nan, 0])
Out[143]:
```

```
0      1
1    NaN
2      2
3    NaN
4      0
5      3
```

В аргументе можно передавать также словарь:

```
In [144]: data.replace({-999: np.nan, -1000: 0})
Out[144]:
0      1
1    NaN
2      2
3    NaN
4      0
5      3
```

Переименование индексов осей

Как и значения в объекте Series, метки осей можно преобразовывать с помощью функции или отображения, порождающего новые объекты с другими метками. Оси можно также модифицировать на месте, не создавая новую структуру данных. Вот простой пример:

```
In [145]: data = DataFrame(np.arange(12).reshape((3, 4)),
   .....:                  index=['Ohio', 'Colorado', 'New York'],
   .....:                  columns=['one', 'two', 'three', 'four'])
```

Как и у объекта Series, у индексов осей имеется метод `map`:

```
In [146]: data.index.map(str.upper)
Out[146]: array([OHIO, COLORADO, NEW YORK], dtype=object)
```

Индексу можно присваивать значение, т. е. модифицировать DataFrame на месте:

```
In [147]: data.index = data.index.map(str.upper)

In [148]: data
Out[148]:
          one  two  three  four
OHIO        0    1      2     3
COLORADO    4    5      6     7
NEW YORK    8    9     10    11
```

Если требуется создать преобразованный вариант набора данных, не меняя оригинал, то будет полезен метод `rename`:

```
In [149]: data.rename(index=str.title, columns=str.upper)
Out[149]:
          ONE  TWO  THREE  FOUR
Ohio        0    1      2     3
```

```
Colorado    4    5      6     7
New York    8    9     10    11
```

Интересно, что `rename` можно использовать в сочетании с похожим на словарь объектом, который возвращает новые значения для подмножества меток оси:

```
In [150]: data.rename(index={'OHIO': 'INDIANA'},
   .....: columns={'three': 'peekaboo'})
Out[150]:
          one  two  peekaboo  four
INDIANA     0    1         2     3
COLORADO    4    5         6     7
NEW YORK    8    9        10    11
```

Метод `rename` избавляет от необходимости копировать объект DataFrame вручную и присваивать значения его атрибутам `index` и `columns`. Чтобы модифицировать набор данных на месте, задайте параметр `inplace=True`:

```
# Всегда возвращает ссылку на DataFrame
In [151]: _ = data.rename(index={'OHIO': 'INDIANA'}, inplace=True)

In [152]: data
Out[152]:
          one  two  three  four
INDIANA     0    1      2     3
COLORADO    4    5      6     7
NEW YORK    8    9     10    11
```

Дискретизация и раскладывание

Непрерывные данные часто дискретизируются или как-то иначе раскладываются по интервалам – «ящикам» – для анализа. Предположим, что имеются данные о группе лиц в каком-то исследовании, и требуется разложить их ящикам, соответствующим возрасту – дискретной величине:

```
In [153]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Разобьем эти ящики на группы: от 18 до 25, от 26 до 35, от 35 до 60 и от 60 и старше. Для этой цели в pandas есть функция `cut`:

```
In [154]: bins = [18, 25, 35, 60, 100]

In [155]: cats = pd.cut(ages, bins)

In [156]: cats
Out[156]:
Categorical:
array([[18, 25], (18, 25], (18, 25], (25, 35], (18, 25], (18, 25],
       (35, 60], (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]], dtype=object)
Levels (4): Index([[18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)
```

pandas возвращает специальный объект `Categorical`. Его можно рассматривать как массив строк с именами ящика; на самом деле он содержит массив `levels`, в

котором хранятся неповторяющиеся имена категорий, а также метки данных ages в атрибуте labels:

```
In [157]: cats.labels
Out[157]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1])

In [158]: cats.levels
Out[158]: Index([(18, 25], (25, 35], (35, 60], (60, 100]], dtype=object)

In [159]: pd.value_counts(cats)
Out[159]:
(18, 25]    5
(35, 60]    3
(25, 35]    3
(60, 100]   1
```

Согласно принятой в математике нотации интервалов круглая скобка означает, что соответствующий конец не включается (*открыт*), а квадратная – что включается (*замкнут*). Чтобы сделать открытым правый конец, следует задать параметр right=False:

```
In [160]: pd.cut(ages, [18, 26, 36, 61, 100], right=False)
Out[160]:
Categorical:
array([[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), [18, 26),
       [36, 61), [26, 36), [61, 100), [36, 61), [36, 61), [26, 36)], dtype=object)
Levels (4): Index([[18, 26), [26, 36), [36, 61), [61, 100)], dtype=object)
```

Можно также самостоятельно задать имена ящиков, передав список или массив в параметре labels:

```
In [161]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [162]: pd.cut(ages, bins, labels=group_names)
Out[162]:
Categorical:
array([Youth, Youth, Youth, YoungAdult, Youth, Youth, MiddleAged,
       YoungAdult, Senior, MiddleAged, MiddleAged, YoungAdult], dtype=object)
Levels (4): Index([Youth, YoungAdult, MiddleAged, Senior], dtype=object)
```

Если передать методу cut целое число ящиков, а не явно заданные границы, то он разобьет данные на группы равной длины, исходя из минимального и максимального значения. Рассмотрим раскладывание равномерно распределенных данных по четырем ящикам:

```
In [163]: data = np.random.rand(20)

In [164]: pd.cut(data, 4, precision=2)
Out[164]:
Categorical:
array([(0.45, 0.67], (0.23, 0.45], (0.0037, 0.23], (0.45, 0.67],
       (0.67, 0.9], (0.45, 0.67], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],
       (0.67, 0.9], (0.67, 0.9], (0.67, 0.9], (0.23, 0.45], (0.23, 0.45],
```

```
       (0.23, 0.45], (0.67, 0.9], (0.0037, 0.23], (0.0037, 0.23],
       (0.23, 0.45], (0.23, 0.45]], dtype=object)
Levels (4): Index([(0.0037, 0.23], (0.23, 0.45], (0.45, 0.67],
                   (0.67, 0.9]], dtype=object)
```

Родственный метод `qcut` раскладывает данные, исходя из выборочных квантилей. Метод `cut` обычно создает ящики, содержащие разное число точек, – это всецело определяемся распределением данных. Но поскольку `qcut` пользуется выборочными квантилями, то по определению получаются ящики равного размера:

```
In [165]: data = np.random.randn(1000) # Normally distributed

In [166]: cats = pd.qcut(data, 4) # Cut into quartiles

In [167]: cats
Out[167]:
Categorical:
array([(-0.022, 0.641], [-3.745, -0.635], (0.641, 3.26], ...,
       (-0.635, -0.022], (0.641, 3.26], (-0.635, -0.022]], dtype=object)
Levels (4): Index([[-3.745, -0.635], (-0.635, -0.022], (-0.022, 0.641],
                   (0.641, 3.26]], dtype=object)

In [168]: pd.value_counts(cats)
Out[168]:
[-3.745, -0.635]  250
(0.641, 3.26]     250
(-0.635, -0.022]  250
(-0.022, 0.641]   250
```

Как и в случае `cut`, можно задать величины квантилей (числа от 0 до 1 включительно) самостоятельно:

```
In [169]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[169]:
Categorical:
array([(-0.022, 1.302], (-1.266, -0.022], (-0.022, 1.302], ...,
       (-1.266, -0.022], (-0.022, 1.302], (-1.266, -0.022]], dtype=object)
Levels (4): Index([[-3.745, -1.266], (-1.266, -0.022], (-0.022, 1.302],
                   (1.302, 3.26]], dtype=object)
```

Мы еще вернемся к методам `cut` и `qcut` в главе об агрегировании и групповых операциях, поскольку эти функции дискретизации особенно полезны для анализа квантилей и групп.

Обнаружение и фильтрация выбросов

Фильтрация или преобразование выбросов – это, в основном, вопрос применения операций с массивами. Рассмотрим объект DataFrame с нормально распределенными данными:

```
In [170]: np.random.seed(12345)

In [171]: data = DataFrame(np.random.randn(1000, 4))

In [172]: data.describe()
```

```
Out[172]:
                0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.067684     0.067924     0.025598    -0.002298
std       0.998035     0.992106     1.006835     0.996794
min      -3.428254    -3.548824    -3.184377    -3.745356
25%      -0.774890    -0.591841    -0.641675    -0.644144
50%      -0.116401     0.101143     0.002073    -0.013611
75%       0.616366     0.780282     0.680391     0.654328
max       3.366626     2.653656     3.260383     3.927528
```

Допустим, что мы хотим найти в одном из столбцов значения, превышающие 3 по абсолютной величине:

```
In [173]: col = data[3]

In [174]: col[np.abs(col) > 3]
Out[174]:
97    3.927528
305  -3.399312
400  -3.745356
Name: 3
```

Чтобы выбрать все строки, в которых встречаются значения, по абсолютной величине превышающие 3, мы можем воспользоваться методом `any` для булева объекта DataFrame:

```
In [175]: data[(np.abs(data) > 3).any(1)]
Out[175]:
             0         1         2         3
5    -0.539741  0.476985  3.248944 -1.021228
97   -0.774363  0.552936  0.106061  3.927528
102  -0.655054 -0.565230  3.176873  0.959533
305  -2.315555  0.457246 -0.025907 -3.399312
324   0.050188  1.951312  3.260383  0.963301
400   0.146326  0.508391 -0.196713 -3.745356
499  -0.293333 -0.242459 -3.056990  1.918403
523  -3.428254 -0.296336 -0.439938 -0.867165
586   0.275144  1.179227 -3.184377  1.369891
808  -0.362528 -3.548824  1.553205 -2.186301
900   3.366626 -2.372214  0.851010  1.332846
```

Можно также присваивать значения данным, удовлетворяющим этому критерию: Следующий код срезает значения, выходящие за границы интервала от –3 до 3:

```
In [176]: data[np.abs(data) > 3] = np.sign(data) * 3

In [177]: data.describe()
Out[177]:
                0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.067684     0.067924     0.025598    -0.002298
```

```
std       0.998035     0.992106     1.006835     0.996794
min      -3.000000    -3.000000    -3.000000    -3.000000
25%      -0.774890    -0.591841    -0.641675    -0.644144
50%      -0.116401     0.101143     0.002073    -0.013611
75%       0.616366     0.780282     0.680391     0.654328
max       3.000000     2.653656     3.000000     3.000000
```

U-функция `np.sign` возвращает массив, содержащий 1 и –1 в зависимости от знака исходного значения.

Перестановки и случайная выборка

Переставить (случайным образом переупорядочить) объект Series или строки объекта DataFrame легко с помощью функции `numpy.random.permutation`. Если передать функции `permutation` длину оси, для которой производится перестановка, то будет возвращен массив целых чисел, описывающий новый порядок:

```
In [178]: df = DataFrame(np.arange(5 * 4).reshape(5, 4))

In [179]: sampler = np.random.permutation(5)

In [180]: sampler
Out[180]: array([1, 0, 2, 3, 4])
```

Этот массив затем можно использовать для индексирования на основе поля `ix` или передать функции `take`:

```
In [181]: df
Out[181]:
    0   1   2   3
0   0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19
```

```
In [182]: df.take(sampler)
Out[182]:
    0   1   2   3
1   4   5   6   7
0   0   1   2   3
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19
```

Чтобы выбрать случайное подмножество без замены, можно, например, вырезать первые `k` элементов массива, возвращенного функцией `permutation`, где `k` – размер требуемого подмножества. Существуют гораздо более эффективные алгоритмы выборки без замены, но это простая стратегия, в которой применяются только уже имеющиеся инструменты:

```
In [183]: df.take(np.random.permutation(len(df))[:3])
Out[183]:
    0   1   2   3
1   4   5   6   7
2   8   9  10  11
3  12  13  14  15
4  16  17  18  19
```

Самый быстрый способ сгенерировать выборку *с заменой* – воспользоваться функцией `np.random.randint`, которая возвращает множество случайных целых чисел:

```
In [184]: bag = np.array([5, 7, -1, 6, 4])

In [185]: sampler = np.random.randint(0, len(bag), size=10)

In [186]: sampler
Out[186]: array([4, 4, 2, 2, 2, 0, 3, 0, 4, 1])

In [187]: draws = bag.take(sampler)

In [188]: draws
Out[188]: array([ 4, 4, -1, -1, -1, 5, 6, 5, 4, 7])
```

Вычисление индикаторных переменных

Еще одно преобразование, часто встречающееся в статистическом моделировании и машинном обучении, – преобразование категориальной переменной в «фиктивную», или «индикаторную» матрицу. Если в столбце объекта DataFrame встречается `k` различных значений, то можно построить матрицу или объект DataFrame с `k` столбцами, содержащими только нули и единицы. В библиотеке pandas для этого имеется функция `get_dummies`, хотя нетрудно написать и свою собственную. Вернемся к приведенному выше примеру DataFrame:

```
In [189]: df = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   .....:                 'data1': range(6)})

In [190]: pd.get_dummies(df['key'])
Out[190]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

Иногда желательно добавить префикс к столбцам индикаторного объекта DataFrame, который затем можно будет слить с другими данными. У функции `get_dummies` для этой цели предусмотрен аргумент `prefix`:

```
In [191]: dummies = pd.get_dummies(df['key'], prefix='key')

In [192]: df_with_dummy = df[['data1']].join(dummies)

In [193]: df_with_dummy
Out[193]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

Если некоторая строка DataFrame принадлежит нескольким категориям, то ситуация немного усложняется. Вернемся к рассмотренному ранее набору данных MovieLens 1M:

```
In [194]: mnames = ['movie_id', 'title', 'genres']

In [195]: movies = pd.read_table('ch07/movies.dat', sep='::', header=None,
   .....:                        names=mnames)

In [196]: movies[:10]
Out[196]:
   movie_id                               title                        genres
0         1                   Toy Story (1995)   Animation|Children's|Comedy
1         2                     Jumanji (1995)  Adventure|Children's|Fantasy
2         3            Grumpier Old Men (1995)                Comedy|Romance
3         4           Waiting to Exhale (1995)                  Comedy|Drama
4         5 Father of the Bride Part II (1995)                       Comedy
5         6                       Heat (1995)         Action|Crime|Thriller
6         7                    Sabrina (1995)                Comedy|Romance
7         8               Tom and Huck (1995)          Adventure|Children's
8         9               Sudden Death (1995)                        Action
9        10                  GoldenEye (1995)     Action|Adventure|Thriller
```

Чтобы добавить индикаторные переменные для каждого жанра, данные придется немного переформатировать. Сначала, построим список уникальных жанров, встречающихся в наборе данных (с помощью элегантного использования `set.union`):

```
In [197]: genre_iter = (set(x.split('|')) for x in movies.genres)

In [198]: genres = sorted(set.union(*genre_iter))
```

Теперь для построения индикаторного DataFrame можно, например, начать с объекта DataFrame, содержащего только нули:

```
In [199]: dummies = DataFrame(np.zeros((len(movies), len(genres))), columns=genres)
```

Затем перебираем все фильмы и присваиваем элементам в каждой строке объекта `dummies` значение 1:

```
In [200]: for i, gen in enumerate(movies.genres):
   .....:     dummies.ix[i, gen.split('|')] = 1
```

После этого можно, как и раньше, соединить с `movies`:

```
In [201]: movies_windic = movies.join(dummies.add_prefix('Genre_'))

In [202]: movies_windic.ix[0]
Out[202]:
movie_id 1
title                              Toy Story (1995)
genres                  Animation|Children's|Comedy
Genre_Action                                     0
Genre_Adventure                                  0
```

```
Genre_Animation                          1
Genre_Children's                         1
Genre_Comedy                             1
Genre_Crime                              0
Genre_Documentary                        0
Genre_Drama                              0
Genre_Fantasy                            0
Genre_Film-Noir                          0
Genre_Horror                             0
Genre_Musical                            0
Genre_Mystery                            0
Genre_Romance                            0
Genre_Sci-Fi                             0
Genre_Thriller                           0
Genre_War                                0
Genre_Western                            0
Name: 0
```

Для очень больших наборов данных такой способ построения индикаторных переменных для нескольких категорий быстрым не назовешь. Но, безусловно, можно написать низкоуровневую функцию, в которой будут использованы знания о внутренней структуре объекта DataFrame.

В статистических приложениях бывает полезно сочетать функцию `get_dummies` с той или иной функцией дискретизации, например `cut`:

```
In [204]: values = np.random.rand(10)

In [205]: values
Out[205]:
array([ 0.9296, 0.3164, 0.1839, 0.2046, 0.5677, 0.5955, 0.9645,
        0.6532, 0.7489, 0.6536])

In [206]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [207]: pd.get_dummies(pd.cut(values, bins))
Out[207]:
   (0, 0.2]  (0.2, 0.4]  (0.4, 0.6]  (0.6, 0.8]  (0.8, 1]
0         0           0           0           0         1
1         0           1           0           0         0
2         1           0           0           0         0
3         0           1           0           0         0
4         0           0           1           0         0
5         0           0           1           0         0
6         0           0           0           0         1
7         0           0           0           1         0
8         0           0           0           1         0
9         0           0           0           1         0
```

Манипуляции со строками

Python уже давно является популярным языком манипулирования данными отчасти потому, что располагает простыми средствами обработки строк и текста.

В большинстве случаев оперировать текстом легко – благодаря наличию встроенных методов у строковых объектов. В более сложных ситуациях, когда нужно сопоставлять текст с образцами, на помощь приходят регулярные выражения. Библиотека pandas расширяет этот инструментарий, позволяя применять методы строк и регулярных выражений к целым массивам и беря на себя возню с отсутствующими значениями.

Методы строковых объектов

Для многих приложений вполне достаточно встроенных методов работы со строками. Например, строку, в которой данные записаны через запятую, можно разбить по поля с помощью метода `split`:

```
In [208]: val = 'a,b, guido'

In [209]: val.split(',')
Out[209]: ['a', 'b', ' guido']
```

Метод `split` часто употребляемся вместе с методом `strip`, чтобы убрать пробельные символы (в том числе перехода на новую строку):

```
In [210]: pieces = [x.strip() for x in val.split(',')]

In [211]: pieces
Out[211]: ['a', 'b', 'guido']
```

Чтобы конкатенировать строки, применяя в качестве разделителя двойное двоеточие, можно использовать оператор сложения:

```
In [212]: first, second, third = pieces

In [213]: first + '::' + second + '::' + third
Out[213]: 'a::b::guido'
```

Но это недостаточно общий метод. Быстрее и лучше соответствует духу Python другой способ: передать список или кортеж методу `join` строки `'::'`:

```
In [214]: '::'.join(pieces)
Out[214]: 'a::b::guido'
```

Существуют также методы для поиска подстрок. Лучше всего искать подстроку с помощью ключевого слова `in`, но методы `index` и `find` тоже годятся:

```
In [215]: 'guido' in val
Out[215]: True

In [216]: val.index(',') In [217]: val.find(':')
Out[216]: 1 Out[217]: -1
```

Разница между `find` и `index` состоит в том, что `index` возбуждает исключение, если строка не найдена (вместо того чтобы возвращать –1):

```
In [218]: val.index(':')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-218-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

Метод `count` возвращает количество вхождений подстроки:

```
In [219]: val.count(',')
Out[219]: 2
```

Метод `replace` заменяет вхождения образца указанной строкой. Он же применяется для удаления подстрок – достаточно в качестве заменяющей передать пустую строку:

```
In [220]: val.replace(',', '::')
Out[220]: 'a::b:: guido'
```

```
In [221]: val.replace(',', '')
Out[221]: 'ab guido'
```

Как мы вскоре увидим, во многих таких операциях можно использовать также регулярные выражения.

Таблица 7.3. Встроенные в Python методы строковых объектов

Метод	Описание
`count`	Возвращает количество неперекрывающихся вхождений подстроки в строку
`endswith`, `startswith`	Возвращает `True`, если строка оканчивается (начинается) указанной подстрокой
`join`	Использовать данную строку как разделитель при конкатенации последовательности других строк
`index`	Возвращает позицию первого символа подстроки в строке. Если подстрока не найдена, возбуждает исключение `ValueError`
`find`	Возвращает позицию первого символа *первого* вхождения подстроки в строку, как и `index`. Но если строка не найдена, то возвращает -1
`rfind`	Возвращает позицию первого символа *последнего* вхождения подстроки в строку. Если строка не найдена, то возвращает -1
`replace`	Заменяет вхождения одной строки другой строкой
`strip`, `rstrip`, `lstrip`	Удаляет пробельные символы, в т. ч. символы новой строки в начале и (или) конце строки.
`split`	Разбивает строку на список подстрок по указанному разделителю
`lower`, `upper`	Преобразует буквы в нижний или верхний регистр соответственно
`ljust`, `rjust`	Выравнивает строку на левую или правую границу соответственно. Противоположный конец строки заполняется пробелами (или каким-либо другим символом), так чтобы получилась строка как минимум заданной длины

Регулярные выражения

Регулярные выражения представляют собой простое средство сопоставления строки с образцом. Синтаксически это строка, записанная с соблюдением правил языка регулярных выражений. Стандартный модуль `re` содержит методы для применения регулярных выражений к строкам, ниже приводятся примеры.

Искусству написания регулярных выражений можно было бы посвятить отдельную главу, но это выходит за рамки данной книги. В Интернете имеется немало отличных пособий и справочных руководств, например, книга Зеда Шоу (Zed Shaw) «Learn Regex The Hard Way» (*http://regex.learncodethehardway.org/book/*).

Функции из модуля `re` можно отнести к трем категориям: сопоставление с образцом, замена и разбиение. Естественно, все они взаимосвязаны; регулярное выражение описывает образец, который нужно найти в тексте, а затем его уже можно применять для разных целей. Рассмотрим простой пример: требуется разбить строку в тех местах, где имеется сколько-то пробельных символов (пробелов, знаков табуляции и знаков новой строки). Для сопоставления с одним или несколькими пробельными символами служит регулярное выражение `\s+`:

```
In [222]: import re

In [223]: text = "foo    bar\t baz  \tqux"

In [224]: re.split('\s+', text)
Out[224]: ['foo', 'bar', 'baz', 'qux']
```

При обращении `re.split('\s+', text)` сначала *компилируется* регулярное выражение, а затем его методу `split` передается заданный текст. Можно просто откомпилировать регулярное выражение, создав тем самым объект, допускающий повторное использование:

```
In [225]: regex = re.compile('\s+')

In [226]: regex.split(text)
Out[226]: ['foo', 'bar', 'baz', 'qux']
```

Чтобы получить список всех подстрок, отвечающих данному регулярному выражению, следует воспользоваться методом `findall`:

```
In [227]: regex.findall(text)
Out[227]: ['    ', '\t ', '  \t']
```

Чтобы не прибегать к громоздкому экранированию знаков `\` в регулярном выражении, пользуйтесь *примитивными* (raw) строковыми литералами, например, `r'C:\x'` вместо `'C:\\x'`.

Создавать объект регулярного выражения с помощью метода `re.compile` рекомендуется, если вы планируете применять одно и то же выражение к нескольким строкам, при этом экономится процессорное время.

С `findall` тесно связаны методы `match` и `search`. Если `findall` возвращает все найденные в строке соответствия, то `search` – только первое. А метод `match` находит *только* соответствие, начинающееся в начале строки. В качестве не столь тривиального примера рассмотрим блок текста и регулярное выражение, распознающее большинство адресов электронной почты:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# Флаг re.IGNORECASE делает регулярное выражение нечувствительным к регистру
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Применение метода `findall` к этому тексту порождает список почтовых адресов:

```
In [229]: regex.findall(text)
Out[229]: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com', 'ryan@yahoo.com']
```

Метод `search` возвращает специальный объект соответствия для первого встретившегося в тексте адреса. В нашем случае этот объект может сказать только о начальной и конечной позиции найденного в строке образца:

```
In [230]: m = regex.search(text)

In [231]: m
Out[231]: <_sre.SRE_Match at 0x10a05de00>

In [232]: text[m.start():m.end()]
Out[232]: 'dave@google.com'
```

Метод `regex.match` возвращает `None`, потому что он находит соответствие образцу только в начале строки:

```
In [233]: print regex.match(text)
None
```

Метод `sub` возвращает новую строку, в которой вхождения образца заменены указанной строкой:

```
In [234]: print regex.sub('REDACTED', text)
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Предположим, что мы хотим найти почтовые адреса и в то же время разбить каждый адрес на три компонента: имя пользователя, имя домена и суффикс домена. Для этого заключим соответствующие части образца в скобки:

```
In [235]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [236]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Метод `groups` объекта соответствия, порожденного таким модифицированным регулярным выражением, возвращает кортеж компонентов образца:

```
In [237]: m = regex.match('wesm@bright.net')

In [238]: m.groups()
Out[238]: ('wesm', 'bright', 'net')
```

Если в образце есть группы, то метод `findall` возвращает список кортежей:

```
In [239]: regex.findall(text)
Out[239]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

Метод `sub` тоже имеет доступ к группам в каждом найденном соответствии с помощью специальных конструкций `\1`, `\2` и т. д.:

```
In [240]: print regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text)
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

О регулярных выражениях в Python можно рассказывать еще долго, но большая часть этого материала выходит за рамки данной книги. Просто для иллюстрации приведу вариант регулярного выражения для распознавания почтовых адресов, в котором группам присваиваются имена:

```
regex = re.compile(r"""
    (?P<username>[A-Z0-9._%+-]+)
    @
    (?P<domain>[A-Z0-9.-]+)
    \.
    (?P<suffix>[A-Z]{2,4})""", flags=re.IGNORECASE|re.VERBOSE)
```

Объект соответствия, порожденный таким регулярным выражением, может генерировать удобный словарь с указанными именами групп:

```
In [242]: m = regex.match('wesm@bright.net')

In [243]: m.groupdict()
Out[243]: {'domain': 'bright', 'suffix': 'net', 'username': 'wesm'}
```

Таблица 7.4. Методы регулярных выражений

Метод	Описание
`findall, finditer`	Возвращает все непересекающиеся образцы, найденные в строке. `findall` возвращает список всех образцов, а `finditer` – итератор, который перебирает их поодиночке
`match`	Ищет соответствие образцу в начале строки и факультативно выделяет в образце группы. Если образец найден, возвращает объект соответствия, иначе `None`
`search`	Ищет в строке образец; если найден, возвращает объект соответствия. В отличие от `match`, образец может находиться в любом месте строки, а не только в начале
`split`	Разбивает строку на части в местах вхождения образца
`sub, subn`	Заменяет все (`sub`) или только первые `n` (`subn`) вхождений образца указанной строкой. Чтобы в указанной строке сослаться на группы, выделенные в образце, используйте конструкции `\1`, `\2`, ...

Векторные строковые функции в pandas

Очистка замусоренного набора данных для последующего анализа подразумевает значительный объем манипуляций со строками и использование регулярных выражений. А чтобы жизнь не казалась медом, в столбцах, содержащих строки, иногда встречаются отсутствующие значения:

```
In [244]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
   .....:         'Rob': 'rob@gmail.com', 'Wes': np.nan}

In [245]: data = Series(data)

In [246]: data                          In [247]: data.isnull()
Out[246]:                               Out[247]:
Dave    dave@google.com                 Dave    False
Rob       rob@gmail.com                 Rob     False
Steve   steve@gmail.com                 Steve   False
Wes                 NaN                 Wes      True
```

Методы строк и регулярных выражений можно применить к каждому значению с помощью метода `data.map` (которому передается лямбда или другая функция), но для отсутствующих значений они «грохнутся». Чтобы справиться этой проблемой, в классе Series есть методы для операций со строками, которые пропускают отсутствующие значения. Доступ к ним производится через атрибут `str`; например, вот как можно было бы с помощью метода `str.contains` проверить, содержит ли каждый почтовый адрес подстроку `'gmail'`:

```
In [248]: data.str.contains('gmail')
Out[248]:
Dave     False
Rob       True
Steve     True
Wes        NaN
```

Регулярные выражения тоже можно так использовать, равно как и их флаги типа IGNORECASE:

```
In [249]: pattern
Out[249]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

In [250]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[250]:
Dave      [('dave', 'google', 'com')]
Rob         [('rob', 'gmail', 'com')]
Steve     [('steve', 'gmail', 'com')]
Wes                               NaN
```

Существует два способа векторной выборки элементов: str.get или доступ к атрибуту str по индексу:

```
In [251]: matches = data.str.match(pattern, flags=re.IGNORECASE)

In [252]: matches
Out[252]:
Dave     ('dave', 'google', 'com')
Rob        ('rob', 'gmail', 'com')
Steve    ('steve', 'gmail', 'com')
Wes                            NaN

In [253]: matches.str.get(1)
Out[253]:
Dave    google
Rob      gmail
Steve    gmail
Wes        NaN

In [254]: matches.str[0]
Out[254]:
Dave    dave
Rob      rob
Steve  steve
Wes      NaN
```

Аналогичный синтаксис позволяет вырезать строки:

```
In [255]: data.str[:5]
Out[255]:
Dave       dave@
Rob        rob@g
Steve      steve
Wes          NaN
```

Таблица 7.5. Векторные методы строковых объектов

Метод	Описание
cat	Поэлементно конкатенирует строки с необязательным разделителем
contains	Возвращает булев массив, показывающий содержит ли каждая строка указанный образец
count	Подсчитывает количество вхождений образца
endswith, startswith	Эквивалентно x.endswith(pattern) или x.startswith(pattern) для каждого элемента
findall	Возвращает список всех вхождений образца для каждой строки

Метод	Описание
`get`	Доступ по индексу ко всем элементам (выбрать `i`-ый элемент)
`join`	Объединяет строки в каждом элементе Series, вставляя между ними указанный разделитель
`len`	Вычисляет длину каждой строки
`lower, upper`	Преобразование регистра; эквивалентно `x.lower()` или `x.upper()` для каждого элемента
`match`	Вызывает `re.match` с указанным регулярным выражением для каждого элемента, возвращает список выделенных групп
`pad`	Дополняет строки пробелами слева, справа или с обеих сторон
`center`	Эквивалентно `pad(side='both')`
`repeat`	Дублирует значения; например, `s.str.repeat(3)` эквивалентно `x * 3` для каждой строки
`replace`	Заменяет вхождения образца указанной строкой
`slice`	Вырезает каждую строку в объекте Series
`split`	Разбивает строки по разделителю или по регулярному выражению
`strip, rstrip, lstrip`	Убирает пробельные символы, в т. ч. знак новой строки, в начале, в конце или с обеих сторон строки; эквивалентно `x.strip()` (и соответственно `rstrip, lstrip`) для каждого элемента

Пример: база данных о продуктах питания министерства сельского хозяйства США

Министерство сельского хозяйства США публикует данные о пищевой ценности продуктов питания. Английский программист Эшли Уильямс (Ashley Williams) преобразовал эту базу данных в формат JSON (*http://ashleyw.co.uk/project/food-nutrient-database*). Записи выглядят следующим образом:

```
{
 "id": 21441,
 "description": "KENTUCKY FRIED CHICKEN, Fried Chicken, EXTRA CRISPY,
Wing, meat and skin with breading",
 "tags": ["KFC"],
 "manufacturer": "Kentucky Fried Chicken",
 "group": "Fast Foods",
 "portions": [
   {
     "amount": 1,
     "unit": "wing, with skin",
     "grams": 68.0
   },
   ...
```

```
],
"nutrients": [
  {
    "value": 20.8,
    "units": "g",
    "description": "Protein",
    "group": "Composition"
  },
  ...
]
}
```

У каждого продукта питания есть ряд идентифицирующих атрибутов и два списка: питательные элементы и размеры порций. Для анализа данные в такой форме подходят плохо, поэтому необходимо их переформатировать.

Скачав архив с указанного адреса и распаковав его, вы затем можете загрузить его в Python-программу с помощью любой библиотеки для работы с JSON. Я воспользуюсь стандартным модулем Python `json`:

```
In [256]: import json

In [257]: db = json.load(open('ch07/foods-2011-10-03.json'))

In [258]: len(db)
Out[258]: 6636
```

Каждая запись в `db` – словарь, содержащий все данные об одном продукте питания. Поле `'nutrients'` – это список словарей, по одному для каждого питательного элемента:

```
In [259]: db[0].keys() In [260]: db[0]['nutrients'][0]
Out[259]: Out[260]:
[u'portions', {u'description': u'Protein',
 u'description', u'group': u'Composition',
 u'tags', u'units': u'g',
 u'nutrients', u'value': 25.18}
 u'group',
 u'id',
 u'manufacturer']

In [261]: nutrients = DataFrame(db[0]['nutrients'])

In [262]: nutrients[:7]
Out[262]:
                     description        group  units    value
0                        Protein  Composition      g    25.18
1              Total lipid (fat)  Composition      g    29.20
2   Carbohydrate, by difference  Composition      g     3.06
3                            Ash        Other      g     3.28
4                         Energy       Energy   kcal   376.00
5                          Water  Composition      g    39.28
6                         Energy       Energy     kJ  1573.00
```

Преобразуя список словарей в DataFrame, можно задать список полей, которые нужно извлекать. Мы ограничимся названием продукта, группой, идентификатором и производителем:

```
In [263]: info_keys = ['description', 'group', 'id', 'manufacturer']

In [264]: info = DataFrame(db, columns=info_keys)

In [265]: info[:5]
Out[265]:
                        description                 group   id  manufacturer
0                   Cheese, caraway  Dairy and Egg Products 1008
1                   Cheese, cheddar  Dairy and Egg Products 1009
2                      Cheese, edam  Dairy and Egg Products 1018
3                      Cheese, feta  Dairy and Egg Products 1019
4  Cheese, mozzarella, part skim milk  Dairy and Egg Products 1028

In [266]: info
Out[266]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
description   6636  non-null values
group         6636  non-null values
id            6636  non-null values
manufacturer  5195  non-null values
dtypes: int64(1), object(3)
```

Метод `value_counts` покажет распределение продуктов питания по группам:

```
In [267]: pd.value_counts(info.group)[:10]
Out[267]:
Vegetables and Vegetable Products    812
Beef Products                        618
Baked Products                       496
Breakfast Cereals                    403
Legumes and Legume Products          365
Fast Foods                           365
Lamb, Veal, and Game Products        345
Sweets                               341
Pork Products                        328
Fruits and Fruit Juices              328
```

Чтобы теперь произвести анализ данных о питательных элементах, проще всего собрать все питательные элементы для всех продуктов в одну большую таблицу. Для этого понадобится несколько шагов. Сначала я преобразую каждый список питательных элементов в объект DataFrame, добавлю столбец id, содержащий идентификатор продукта, и помещу этот DataFrame в список. После этого все объекты можно будет конкатенировать методом `concat`:

```
nutrients = []

for rec in db:
```

```
    fnuts = DataFrame(rec['nutrients'])
    fnuts['id'] = rec['id']
    nutrients.append(fnuts)

nutrients = pd.concat(nutrients, ignore_index=True)
```

Если все пройдет хорошо, то объект `nutrients` будет выглядеть следующим образом:

```
In [269]: nutrients
Out[269]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 389355 entries, 0 to 389354
Data columns:
description  389355  non-null values
group        389355  non-null values
units        389355  non-null values
value        389355  non-null values
id           389355  non-null values
dtypes: float64(1), int64(1), object(3)
```

Я заметил, что по какой-то причине в этом DataFrame есть дубликаты, поэтому лучше их удалить:

```
In [270]: nutrients.duplicated().sum()

Out[270]: 14179
In [271]: nutrients = nutrients.drop_duplicates()
```

Поскольку столбцы `'group'` и `'description'` есть в обоих объектах DataFrame, переименуем их, чтобы было понятно, что есть что:

```
In [272]: col_mapping = {'description' : 'food',
   .....:                'group' : 'fgroup'}

In [273]: info = info.rename(columns=col_mapping, copy=False)

In [274]: info
Out[274]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6636 entries, 0 to 6635
Data columns:
food          6636  non-null values
fgroup        6636  non-null values
id            6636  non-null values
manufacturer  5195  non-null values
dtypes: int64(1), object(3)

In [275]: col_mapping = {'description' : 'nutrient',
   .....:                'group' : 'nutgroup'}

In [276]: nutrients = nutrients.rename(columns=col_mapping, copy=False)

In [277]: nutrients
Out[277]:
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 389354
Data columns:
nutrient  375176  non-null values
nutgroup  375176  non-null values
units     375176  non-null values
value     375176  non-null values
id        375176  non-null values
dtypes: float64(1), int64(1), object(3)
```

Сделав все это, мы можем слить `info` с `nutrients`:

```
In [278]: ndata = pd.merge(nutrients, info, on='id', how='outer')

In [279]: ndata
Out[279]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 375176 entries, 0 to 375175
Data columns:
nutrient      375176  non-null values
nutgroup      375176  non-null values
units         375176  non-null values
value         375176  non-null values
id            375176  non-null values
food          375176  non-null values
fgroup        375176  non-null values
manufacturer  293054  non-null values
dtypes: float64(1), int64(1), object(6)

In [280]: ndata.ix[30000]
Out[280]:
nutrient                 Folic acid
nutgroup                   Vitamins
units                           mcg
value                             0
id                             5658
food     Ostrich, top loin, cooked
fgroup             Poultry Products
manufacturer
Name: 30000
```

Средства, необходимые для формирования продольных и поперечных срезов, агрегирования и визуализации этого набора данных, будут подробно рассмотрены в следующих двух главах; познакомившись с ними, вы сможете вернуться к этому набору. Для примера можно было бы построить график медианных значений по группе и типу питательного элемента (рис. 7.1):

```
In [281]: result = ndata.groupby(['nutrient', 'fgroup'])['value'].quantile(0.5)

In [282]: result['Zinc, Zn'].order().plot(kind='barh')
```

Проявив смекалку, вы сможете найти, какой продукт питания наиболее богат каждым питательным элементов:

```
by_nutrient = ndata.groupby(['nutgroup', 'nutrient'])

get_maximum = lambda x: x.xs(x.value.idxmax())
get_minimum = lambda x: x.xs(x.value.idxmin())

max_foods = by_nutrient.apply(get_maximum)[['value', 'food']]

# Немного уменьшить продукт питания
max_foods.food = max_foods.food.str[:50]
```

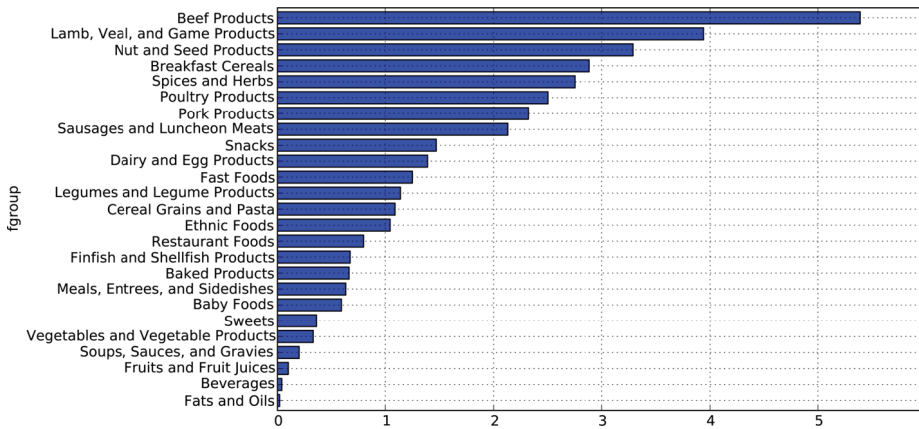


Рис. 7.1. Медианные значения цинка по группе питательных элементов

Получившийся объект DataFrame слишком велик для того, чтобы приводить его полностью. Ниже приведена только группа питательных элементов `'Amino Acids'` (аминокислоты):

```
In [284]: max_foods.ix['Amino Acids']['food']
Out[284]:
nutrient
Alanine                                 Gelatins, dry powder, unsweetened
Arginine                                    Seeds, sesame flour, low-fat
Aspartic acid                                     Soy protein isolate
Cystine                       Seeds, cottonseed flour, low fat (glandless)
Glutamic acid                                     Soy protein isolate
Glycine                                 Gelatins, dry powder, unsweetened
Histidine                        Whale, beluga, meat, dried (Alaska Native)
Hydroxyproline          KENTUCKY FRIED CHICKEN, Fried Chicken, ORIGINAL R
Isoleucine              Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Leucine                 Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Lysine                  Seal, bearded (Oogruk), meat, dried (Alaska Nativ
Methionine                        Fish, cod, Atlantic, dried and salted
Phenylalanine           Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Proline                                 Gelatins, dry powder, unsweetened
Serine                  Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Threonine               Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Tryptophan               Sea lion, Steller, meat with fat (Alaska Native)
```

```
Tyrosine                  Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Valine                    Soy protein isolate, PROTEIN TECHNOLOGIES INTERNA
Name: food
```

ГЛАВА 8. Построение графиков и визуализация

Построение графиков, а также статическая или интерактивная визуализация – одни из важнейших задач анализа данных. Они могут быть частью процесса исследования, например, применяться для выявления выбросов, определения необходимых преобразований данных или поиска идей для построения моделей. В других случаях построение интерактивной визуализации для веб-сайта, например с помощью библиотеки d3.js (*http://d3js.org/*), может быть конечной целью. Для Python имеется много инструментов визуализации (см. перечень в конце этой главы), но я буду использовать в основном matplotlib (*http://matplotlib.sourceforge.net*).

Matplotlib – это пакет для построения графиков (главным образом, двумерных) полиграфического качества. Проект был основан Джоном Хантером в 2002 году с целью реализовать на Python интерфейс, аналогичный MATLAB. Впоследствии он сам, Фернандо Перес (из группы разработчиков IPython) и другие люди в течение многих лет работали над тем, чтобы сделать комбинацию IPython и matplotlib максимально функциональной и продуктивной средой для научных расчетов. При использовании в сочетании с какой-нибудь библиотекой ГИП (например, внутри IPython), matplotlib приобретает интерактивные возможности: панорамирование, масштабирование и другие. Этот пакет поддерживает разнообразные системы ГИП во всех операционных системах, а также умеет экспортировать графические данные во всех векторных и растровых форматах: PDF, SVG, JPG, PNG, BMP, GIF и т. д. С его помощью я построил почти все рисунки для этой книги.

Для matplotlib имеется целый ряд дополнительных библиотек, например `mplot3d` для построения трехмерных графиков и `basemap` для построения карт и проекций. В конце главе я приведу пример использования `basemap` для нанесения данных на карту и чтения *shape-файлов*. Для проработки приведенных в этой главе примеров код не забудьте загрузить IPython в режиме pylab (`ipython --pylab`) или включить интеграцию с циклом обработки событий ГИП с помощью магической функции `%gui`.

Краткое введение в API библиотеки matplotlib

Взаимодействовать с matplotlib можно несколькими способами. Самый распространенный – запустить IPython в *режиме pylab* с помощью команды `ipython --pylab`. В результате IPython конфигурируется для поддержки выбранной системы ГИП (Tk, wxPython, PyQt, платформенный ГИП Mac OS X, GTK). Для большинства пользователей подразумеваемой по умолчанию системы ГИП достаточно. В режиме pylab в IPython также импортируется много модулей и функций, чтобы интерфейс был больше похож на MATLAB. Убедиться, что все работает, можно, построив простой график:

```
plot(np.arange(10))
```

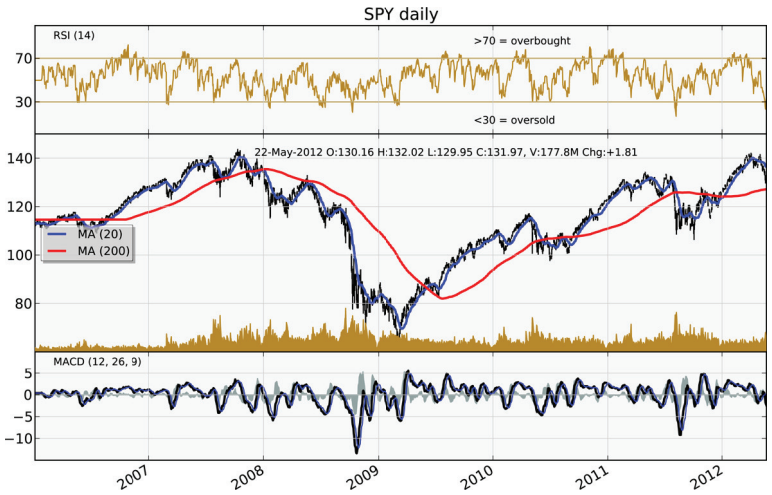


Рис. 8.1. Более сложный финансовый график, построенный matplotlib

Если все настроено правильно, то появится новое окно с линейным графиком. Окно можно закрыть мышью или введя команду `close()`. Все функции matplotlib API, в частности `plot` и `close`, находятся в модуле `matplotlib.pyplot`, при импорте которого обычно придерживаются следующего соглашения:

```
import matplotlib.pyplot as plt
```

В этой книге не хватит места для рассмотрения функциональности matplotlib во всей полноте. Но достаточно показать, что делать, и дальше вы все освоите самостоятельно. Для того чтобы стать настоящим экспертом в построении графиков, нет ничего лучше галереи и документации matplotlib.

Описываемые ниже функции построения графиков из библиотеки pandas берут на себя многие рутинные детали, но если предусмотренных в них параметров вам недостаточно, то придется разбираться с matplotlib API.

Рисунки и подграфики

Графики в matplotlib «живут» внутри объекта рисунка `Figure`. Создать новый рисунок можно методом `plt.figure`:

```
In [13]: fig = plt.figure()
```

Если вы работаете в режиме pylab в IPython, то должно появиться новое пустое окно. У метода `plt.figure` много параметров, в частности `figsize` гарантирует, что при сохранении рисунка на диске у него будут определенные размер и отношение сторон. Рисунки в matplotlib поддерживают схему нумерации (например, `plt.figure(2)`) в подражание MATLAB. Для получения ссылки на активный рисунок служит метод `plt.gcf()`.

Нельзя создать график, имея пустой рисунок. Сначала нужно создать один или несколько подграфиков с помощью метода `add_subplot`:

```
In [14]: ax1 = fig.add_subplot(2, 2, 1)
```

Это означает, что рисунок будет расчерчен сеткой 2 × 2, и мы выбираем первый из четырех подграфиков (нумерация начинается с 1). Если создать следующие два подграфика, то получится рисунок, изображенный на рис. 8.2.

```
In [15]: ax2 = fig.add_subplot(2, 2, 2)

In [16]: ax3 = fig.add_subplot(2, 2, 3)
```

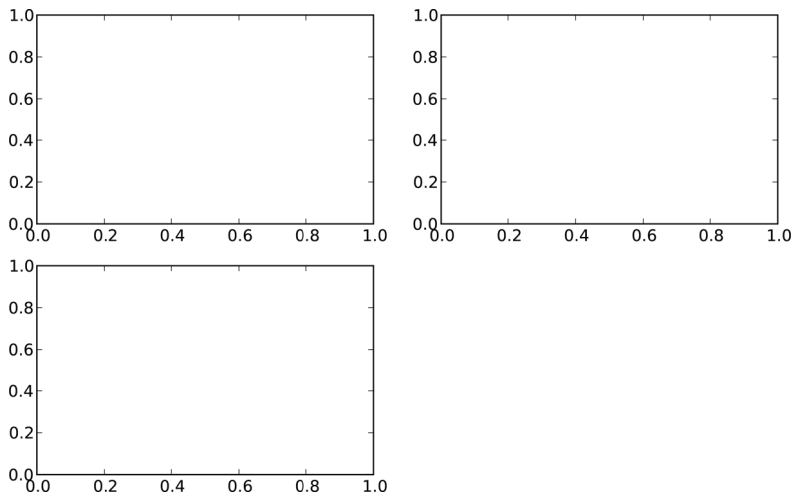


Рис. 8.2. Пустой рисунок matplotlib с тремя подграфиками

При выполнении команды построения графика, например `plt.plot([1.5, 3.5, -2, 1.6])`, matplotlib использует последний созданный рисунок и подграфик (при необходимости создав то и другое) и тем самым маскирует создание рисунка и подграфика. Следовательно, выполнив показанную ниже команду, мы получим картину, изображенную на рис. 8.3:

```
In [17]: from numpy.random import randn
In [18]: plt.plot(randn(50).cumsum(), 'k--')
```

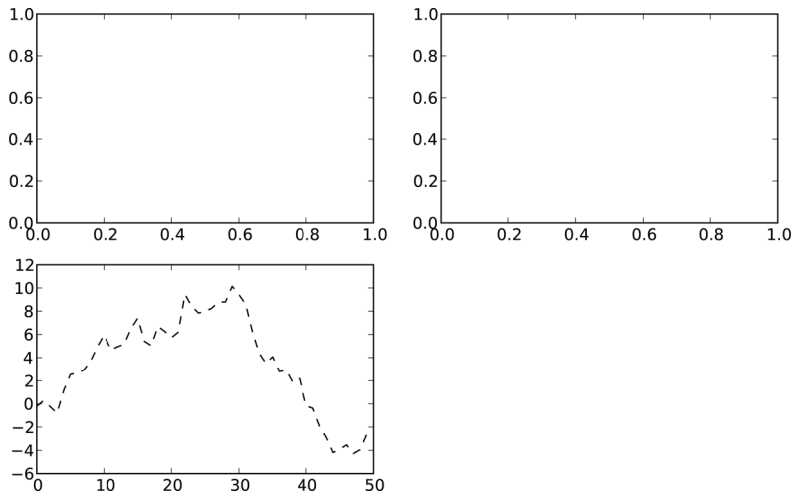


Рис. 8.3. Рисунок после построения одного графика

Параметр *стиля* `'k--'` говорит matplotlib, что график нужно рисовать черной штриховой линией. Метод `fig.add_subplot` возвращает объект `AxesSubplot`, что позволяет рисовать в любом подграфике, вызывая методы этого объекта (см. рис. 8.4):

```
In [19]: _ = ax1.hist(randn(100), bins=20, color='k', alpha=0.3)
In [20]: ax2.scatter(np.arange(30), np.arange(30) + 3 * randn(30))
```

Полный перечень типов графиков имеется в документации по matplotlib. Поскольку создание рисунка с несколькими подграфиками, расположенными определенным образом, – типичная задача, существует вспомогательный метод `plt.subplots`, который создает новый рисунок и возвращает массив NumPy, содержащий созданные в нем объекты подграфиков:

```
In [22]: fig, axes = plt.subplots(2, 3)

In [23]: axes
Out[23]:
array([[Axes(0.125,0.536364;0.227941x0.363636),
        Axes(0.398529,0.536364;0.227941x0.363636),
        Axes(0.672059,0.536364;0.227941x0.363636)],
```

```
[Axes(0.125,0.1;0.227941x0.363636),
 Axes(0.398529,0.1;0.227941x0.363636),
 Axes(0.672059,0.1;0.227941x0.363636)]], dtype=object)
```

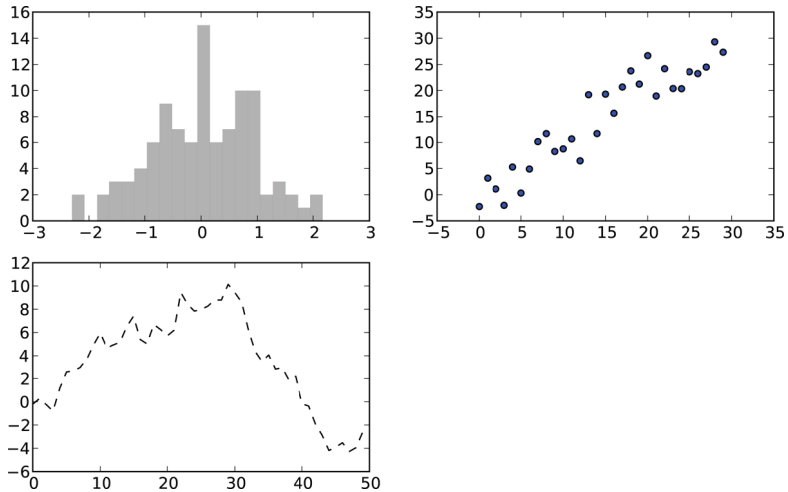


Рис. 8.4. Рисунок после построения дополнительных графиков

Это очень полезно, потому что к массиву `axes` вполне можно обращаться как к двумерному массиву, например `axes[0, 1]`. Можно также указать, что подграфики должны иметь общую ось X или Y, задав параметры `sharex` и `sharey` соответственно. Особенно это удобно, когда надо сравнить данные в одном масштабе; иначе matplotlib автоматически и независимо выбирает масштаб графика. Подробнее об этом методе см. табл. 8.1.

Таблица 8.1. Параметры метода pyplot.subplots

Аргумент	Описание
`nrows`	Число строк в сетке подграфиков
`ncols`	Число столбцов в сетке подграфиков
`sharex`	Все подграфики должны иметь одинаковые риски на оси X (настройка `xlim` отражается на всех подграфиках)
`sharey`	Все подграфики должны иметь одинаковые риски на оси Y (настройка `ylim` отражается на всех подграфиках)
`subplot_kw`	Словарь ключевых слов для создания подграфиков
`**fig_kw`	Дополнительные ключевые слова используются при создании рисунка, например, `plt.subplots(2, 2,figsize=(8, 6))`

Задание свободного места вокруг подграфиков

По умолчанию matplotlib оставляет пустое место вокруг каждого подграфика и между подграфиками. Размер этого места определяется относительно высоты

и ширины графика, так что если изменить размер графика программно или вручную (изменив размер окна), то график автоматически перестроится. Величину промежутка легко изменить с помощью метода `subplots_adjust` объекта `Figure`, который также доступен в виде функции верхнего уровня:

```
subplots_adjust(left=None, bottom=None, right=None, top=None,
                wspace=None, hspace=None)
```

Параметры `wspace` и `hspace` определяют, какой процент от ширины (соответственно высоты) рисунка должен составлять промежуток между подграфиками. В примере ниже я задал нулевой промежуток (рис. 8.5):

```
fig, axes = plt.subplots(2, 2, sharex=True, sharey=True)
for i in range(2):
  for j in range(2):
     axes[i, j].hist(randn(500), bins=50, color='k', alpha=0.5)
plt.subplots_adjust(wspace=0, hspace=0)
```

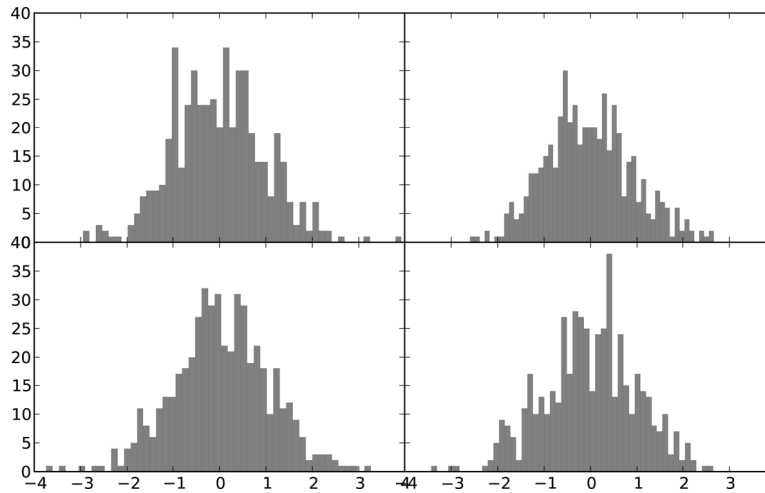


Рис. 8.5. Рисунок, в котором подграфики не разделены промежутками

Вы, наверное, заметили, что риски на осях наложились друг на друга. matplotlib это не проверяет, поэтому если такое происходит, то вам придется самостоятельно подкорректировать риски, явно указав их положения и надписи. Подробнее об этом будет рассказано в следующих разделах.

Цвета, маркеры и стили линий

Главная функция matplotlib – `plot` – принимает массивы координат X и Y, а также необязательную строку, в которой закодированы цвет и стиль линии. Например, чтобы нарисовать график зависимости `y` от `x` зеленой штриховой линией, нужно выполнить следующий вызов:

```
ax.plot(x, y, 'g--')
```

Такой способ задания цвета и стиля линий в виде строки – не более чем удобство; на практике, когда графики строятся из программы, лучше не запутывать код строковыми обозначениями стиля. Этот график можно было бы описать и более понятно:

```
ax.plot(x, y, linestyle='--', color='g')
```

Существует ряд сокращений для наиболее употребительных цветов, но вообще любой цвет можно представить своим RGB-значением (например, `'#CECECE'`). Полный перечень стилей линий имеется в строке документации для функции `plot`. Линейные графики могут быть также снабжены *маркерами*, обозначающими точки, по которым построен график. Поскольку matplotlib создает непрерывный линейный график, производя интерполцию между точками, иногда неясно, где же находятся исходные точки. Маркер можно задать в строке стиля: сначала цвет, потом тип маркера и в конце стиль линии (рис. 8.6):

```
In [28]: plt.plot(randn(30).cumsum(), 'ko--')
```

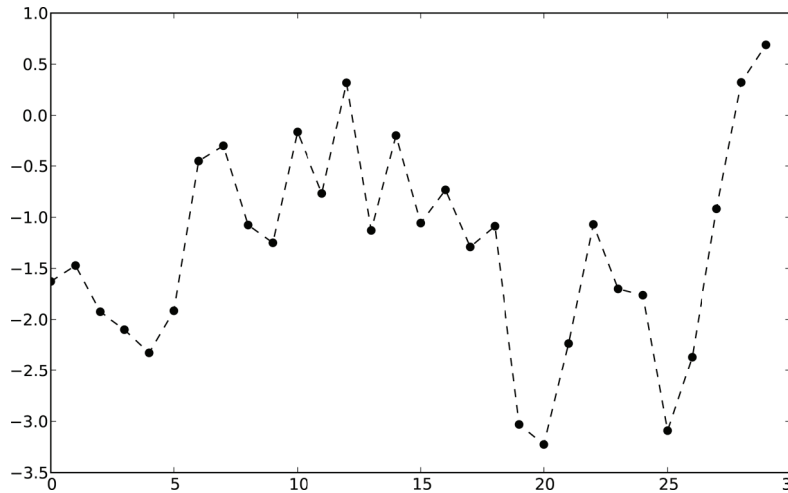


Рис. 8.6. Линейный график с примером маркеров

То же самое можно записать явно:

```
plot(randn(30).cumsum(), color='k', linestyle='dashed', marker='o')
```

По умолчанию на линейных графиках соседние точки соединяются отрезками прямой, т. е. производится линейная интерполяция. Параметр `drawstyle` позволяет изменить этот режим:

```
In [30]: data = randn(30).cumsum()

In [31]: plt.plot(data, 'k--', label='Default')
```

```
Out[31]: [<matplotlib.lines.Line2D at 0x461cdd0>]

In [32]: plt.plot(data, 'k-', drawstyle='steps-post', label='steps-post')
Out[32]: [<matplotlib.lines.Line2D at 0x461f350>]

In [33]: plt.legend(loc='best')
```

Риски, метки и надписи

Для оформления большинства графиков существуют два основных способа: процедурный интерфейс `pyplot` (который будет понятен пользователям MATLAB) и собственный объектно-ориентированный matplotlib API.

Интерфейс `pyplot`, предназначенный для интерактивного использования, состоит из методов `xlim`, `xticks` и `xticklabels`. Они управляют размером области, занятой графиком, положением и метками рисок соответственно. Использовать их можно двумя способами.

- При вызове без аргументов возвращается текущее значение параметра. Например, метод `plt.xlim()` возвращает текущий диапазон значений по оси X.

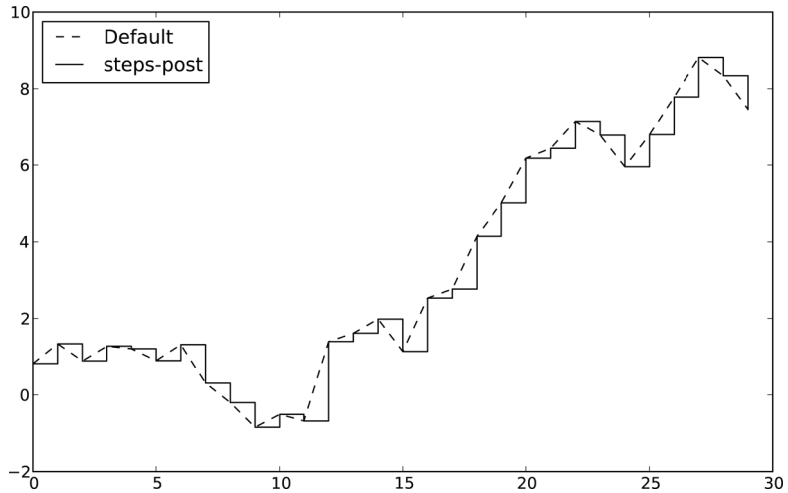


Рис. 8.7. Линейный график с различными значениями параметра `drawstyle`

- При вызове с аргументами устанавливается новое значение параметра. Например, в результате вызова `plt.xlim([0, 10])` диапазон значений по оси X устанавливается от 0 до 10.

Все подобные методы действуют на активный или созданный последним объект `AxesSubplot`. Каждому из них соответствуют два метода самого объекта подграфика; в случае `xlim` это методы `ax.get_xlim` и `ax.set_xlim`. Я предпочитаю пользоваться методами экземпляра подграфика, чтобы код получался понятнее (в особенности, когда работаю с несколькими подграфиками), но вы, конечно, вольны выбирать то, что вам больше нравится.

Задание названия графика, названий осей, рисок и их меток

Чтобы проиллюстрировать оформление осей, я создам простой рисунок и в нем график случайного блуждания (рис. 8.8):

```
In [34]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [35]: ax.plot(randn(1000).cumsum())
```

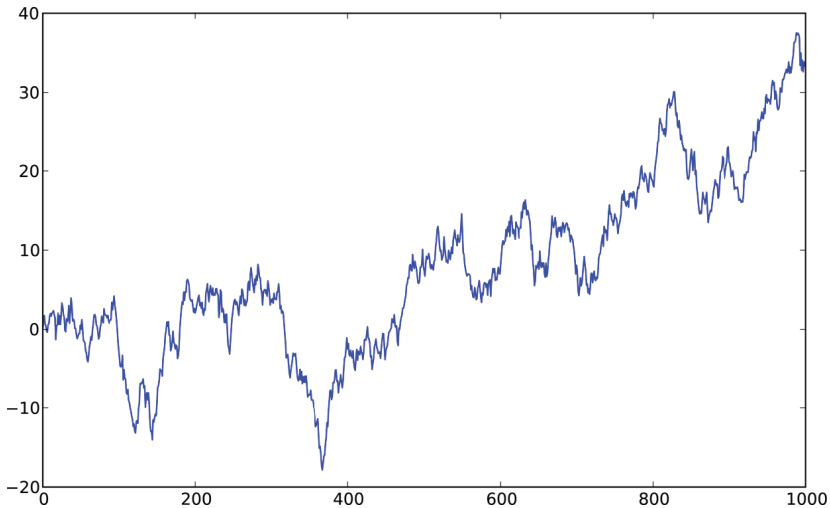


Рис. 8.8. Простой график для иллюстрации рисок

Для изменения рисок на оси X проще всего воспользоваться методами `set_xticks` и `set_xticklabels`. Первый говорит matplotlib, где в пределах диапазона значений данных ставить риски; по умолчанию их числовые значения изображаются также и в виде меток. Но можно задать и другие метки с помощью метода `set_xticklabels`:

```
In [36]: ticks = ax.set_xticks([0, 250, 500, 750, 1000])
In [37]: labels = ax.set_xticklabels(['one', 'two', 'three', 'four', 'five'],
   ....:                             rotation=30, fontsize='small')
```

Наконец, метод `set_xlabel` именует ось X, а метод `set_title` задает название подграфика:

```
In [38]: ax.set_title('My first matplotlib plot')
Out[38]: <matplotlib.text.Text at 0x7f9190912850>

In [39]: ax.set_xlabel('Stages')
```

Получившийся рисунок изображен на рис. 8.9. Для изменения рисок на оси Y нужно сделать то же самое, что и выше, с заменой `x` на `y`.

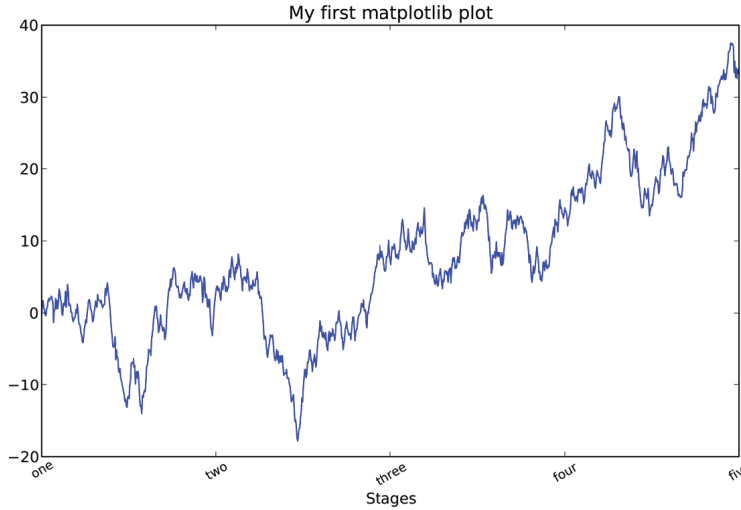


Рис. 8.9. Простой график для иллюстрации рисок

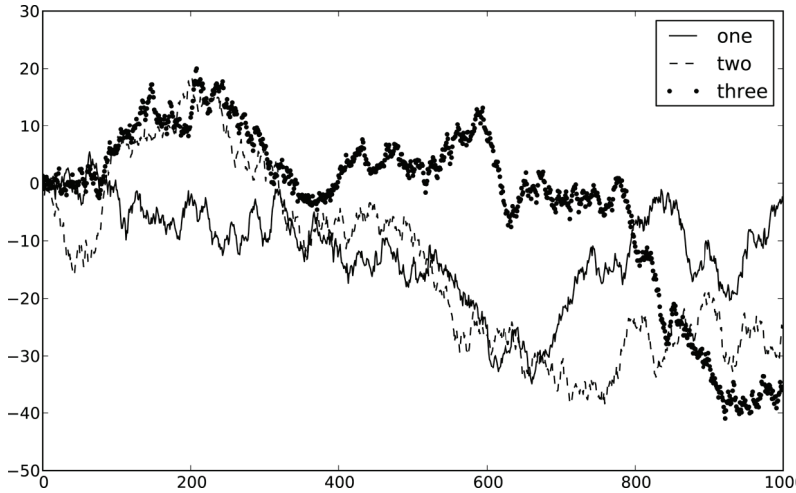


Рис. 8.10. Простой график с тремя линиями и пояснительной надписью

Добавление пояснительных надписей

Пояснительная надпись – еще один важный элемент оформления графика. Добавить ее можно двумя способами. Проще всего передать аргумент `label` при добавлении каждого нового графика:

```
In [40]: fig = plt.figure(); ax = fig.add_subplot(1, 1, 1)

In [41]: ax.plot(randn(1000).cumsum(), 'k', label='one')
```

```
Out[41]: [<matplotlib.lines.Line2D at 0x4720a90>]

In [42]: ax.plot(randn(1000).cumsum(), 'k--', label='two')
Out[42]: [<matplotlib.lines.Line2D at 0x4720f90>]

In [43]: ax.plot(randn(1000).cumsum(), 'k.', label='three')
Out[43]: [<matplotlib.lines.Line2D at 0x4723550>]
```

После этого можно вызвать метод `ax.legend()` или `plt.legend()`, и он автоматически создаст пояснительную надпись:

```
In [44]: ax.legend(loc='best')
```

См. рис. 8.10. Параметр `loc` говорит, где поместить надпись. Если вам все равно, задавайте значение `'best'`, потому что тогда место будет выбрано так, чтобы по возможности не загораживать сам график. Чтобы исключить из надписи один или несколько элементов, не задавайте параметр `label` вовсе или задайте `label='_nolegend_'`.

Аннотации и рисование в подграфике

Помимо стандартных типов графиков разрешается наносить на график свои аннотации, которые могут включать текст, стрелки и другие фигуры.

Для добавления аннотаций и текста предназначены функции `text`, `arrow` и `annotate`. Функция `text` наносит на график текст, начиная с точки с заданными координатами `(x, y)`, с факультативной стилизацией:

```
ax.text(x, y, 'Hello world!',
        family='monospace', fontsize=10)
```

В аннотациях могут встречаться текст и стрелки. В качестве примера построим график цен закрытия по индексу S&P 500, начиная с 2007 года (данные получены с сайта Yahoo! Finance) и аннотируем его некоторыми важными датами, относящимися к финансовому кризису 2008–2009 годов. Результат изображен на рис. 8.11.

```
from datetime import datetime

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

data = pd.read_csv('ch08/spx.csv', index_col=0, parse_dates=True)
spx = data['SPX']

spx.plot(ax=ax, style='k-')

crisis_data = [
  (datetime(2007, 10, 11), 'Peak of bull market'),
  (datetime(2008, 3, 12), 'Bear Stearns Fails'),
  (datetime(2008, 9, 15), 'Lehman Bankruptcy')
```

```
]

for date, label in crisis_data:
  ax.annotate(label, xy=(date, spx.asof(date) + 50),
              xytext=(date, spx.asof(date) + 200),
              arrowprops=dict(facecolor='black'),
              horizontalalignment='left', verticalalignment='top')

# Оставить только диапазон 2007-2010
ax.set_xlim(['1/1/2007', '1/1/2011'])
ax.set_ylim([600, 1800])

ax.set_title('Important dates in 2008-2009 financial crisis')
```

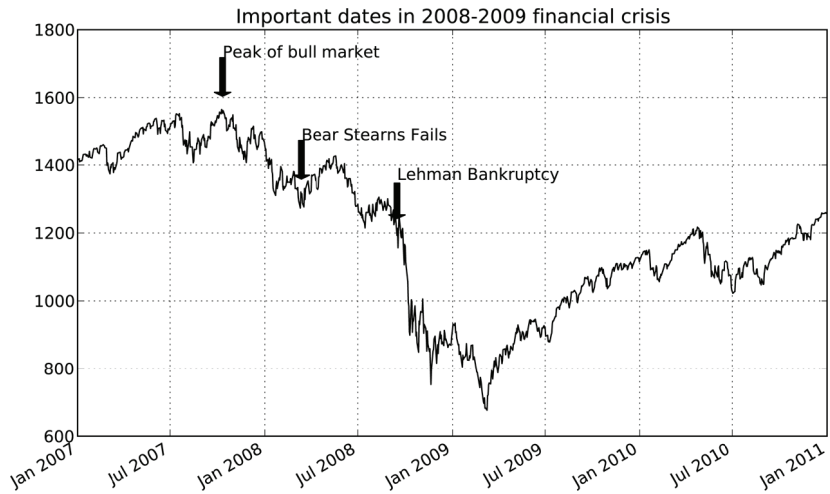


Рис. 8.11. Важные даты, относящиеся к финансовому кризису 2008–2009 годов

В галерее matplotlib в сети есть много других поучительных примеров аннотаций.

Для рисования фигур требуется больше усилий. В matplotlib имеются объекты, соответствующие многим стандартным фигурам, они называются *патчами* (patches). Часть из них, например `Rectangle` и `Circle`, находится в модуле `matplotlib.pyplot`, а весь набор – в модуле `matplotlib.patches`.

Чтобы поместить на график фигуру, мы создаем объект патча `shp` и добавляем его в подграфик, вызывая метод `ax.add_patch(shp)` (см. рис. 8.12):

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)

rect = plt.Rectangle((0.2, 0.75), 0.4, 0.15, color='k', alpha=0.3)
circ = plt.Circle((0.7, 0.2), 0.15, color='b', alpha=0.3)
pgon = plt.Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
                   color='g', alpha=0.5)

ax.add_patch(rect)
```

```
ax.add_patch(circ)
ax.add_patch(pgon)
```

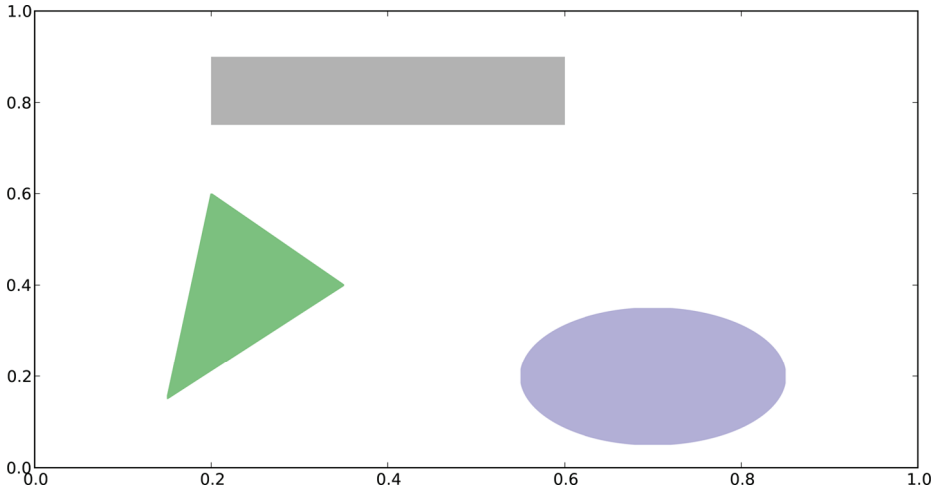


Рис. 8.12. Рисунок, составленный из трех разных фигур

Заглянув в код многих знакомых типов графиков, вы увидите, что они составлены из патчей.

Сохранение графиков в файле

Активный рисунок можно сохранить в файле методом `plt.savefig`. Этот метод эквивалентен методу экземпляра рисунка `savefig`. Например, чтобы сохранить рисунок в формате SVG, достаточно указать только имя файла:

```
plt.savefig('figpath.svg')
```

Формат выводится из расширения имени файла. Если бы мы задали файл с расширением `.pdf`, то рисунок был бы сохранен в формате PDF. При публикации графики я часто использую два параметра: `dpi` (разрешение в точках на дюйм) и `bbox_inches` (размер пустого места вокруг рисунка). Чтобы получить тот же самый график в формате PNG с минимальным обрамлением и разрешением 400 DPI, нужно было бы написать:

```
plt.savefig('figpath.png', dpi=400, bbox_inches='tight')
```

Метод `savefig` может писать не только на диск, а в любой похожий на файл объект, например `StringIO`:

```
from io import StringIO
buffer = StringIO()
plt.savefig(buffer)
plot_data = buffer.getvalue()
```

Например, это полезно для публикации динамически сгенерированных изображений в веб:

Таблица 8.2. Параметры метода Figure.savefig

Аргумент	Описание
`fname`	Строка, содержащая путь к файлу или похожий на файл объект Python. Формат рисунка определяется по расширению имени файла, например: PDF для .pdf и PNG для .png
`dpi`	Разрешение рисунка в точках на дюйм; по умолчанию 100, но может настраиваться
`facecolor, edgecolor`	Цвет фона рисунка вне области, занятой подграфиками. По умолчанию `'w'` (белый)
`format`	Явно заданный формат файла (`'png'`, `'pdf'`, `'svg'`, `'ps'`, `'eps'` и т. д.)
`bbox_inches`	Какую часть рисунка сохранять. Если задано значение `'tight'`, то метод пытается обрезать все пустое место вокруг рисунка.

Конфигурирование matplotlib

В начальной конфигурации matplotlib заданы цветовые схемы и умолчания, ориентированные главным образом на подготовку рисунков к публикации. По счастью, почти все аспекты поведения по умолчанию можно сконфигурировать с помощью обширного набора глобальных параметров, определяющих размер рисунка, промежутки между подграфиками, цвета, размеры шрифтов, стили сетки и т. д. Есть два основных способа работы с системой конфигурирования matplotlib. Первый – программный, с помощью метода `rc`. Например, чтобы глобально задать размер рисунка равным 10 × 10, нужно написать:

```
plt.rc('figure', figsize=(10, 10))
```

Первый аргумент `rc` – настраиваемый компонент, например: `'figure'`, `'axes'`, `'xtick'`, `'ytick'`, `'grid'`, `'legend'` и т. д. Вслед за ним идут позиционные аргументы, задающие параметры этого компонента. В программе описывать параметры проще всего в виде словаря:

```
font_options = {'family' : 'monospace',
                'weight' : 'bold',
                'size' : 'small'}
plt.rc('font', **font_options)
```

Если требуется более обширная настройка, то можно воспользоваться входящим в состав matplotlib конфигурационным файлом `matplotlibrc` в каталоге `matplotlib/mpl-data`, где перечислены все параметры. Если вы настроите этот файл и поместите его в свой домашний каталог под именем `.matplotlibrc`, то он будет загружаться при каждом использовании matplotlib.

Функции построения графиков в pandas

Как вы могли убедиться, matplotlib – библиотека довольно низкого уровня. График в ней составляется из базовых компонентов: способ отображения данных (тип графика: линейный график, столбчатая диаграмма, коробчатая диаграмма, диаграмма рассеяния, контурный график и т. д.), пояснительная надпись, название, метки рисок и прочие аннотации. Отчасти так сделано потому, что во многих случаях данные, необходимые для построения полного графика, разбросаны по разным объектам. В библиотеке pandas у нас уже есть метки строк, метки столбцов и, возможно, информация о группировке. Это означает, что многие графики, для построения которых средствами matplotlib пришлось бы писать много кода, в pandas могут быть построены с помощью одного-двух коротких предложений. Поэтому в pandas имеется много высокоуровневых методов построения графиков для стандартных типов визуализации, в которых используется информация о внутренней организации объектов DataFrame.

На момент написания книги функциональность построения графиков в pandas пересматривается. Во время программы Google Summer of Code 2012 один студент посвятил все свое время добавлению новых функций и попыткам сделать интерфейс более последовательным и удобным для использования. Поэтому не исключено, что код этого раздела устареет быстрее, чем весь остальной, приведенный в книге. В таком случае вашим лучшим советчиком будет документация по pandas в сети.

Линейные графики

У объектов Series и DataFrame имеется метод `plot`, который умеет строить графики разных типов. По умолчанию он строит линейные графики (см. рис. 8.13):

```
In [55]: s = Series(np.random.randn(10).cumsum(), index=np.arange(0, 100, 10))

In [56]: s.plot()
```

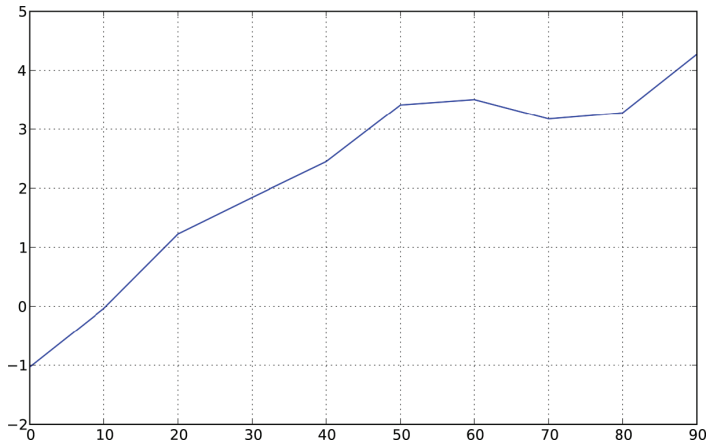


Рис. 8.13. Простой пример графика для объекта Series

Индекс объекта Series передается matplotlib для нанесения рисок на ось X, но это можно отключить, задав параметр `use_index=False`. Риски и диапазон значений на оси X можно настраивать с помощью параметров `xticks` и `xlim`, а на оси Y – с помощью параметров `yticks` и `ylim`. Полный перечень параметров метода `plot` приведен в табл. 8.3. О некоторых я расскажу в этом разделе, а остальные оставлю вам для самостоятельного изучения.

Таблица 8.3. Параметры метода Series.plot

Аргумент	Описание
`label`	Метка для пояснительной надписи на графике
`ax`	Объект подграфика matplotlib, внутри которого строить график. Если параметр не задан, то используется активный подграфик
`style`	Строка стиля, например `'ko--'`, которая передается matplotlib
`alpha`	Уровень непрозрачности графика (число от 0 до 1)
`kind`	Может принимать значения `'line'`, `'bar'`, `'barh'`, `'kde'`
`logy`	Использовать логарифмический масштаб по оси Y
`use_index`	Брать метки рисок из индекса объекта
`rot`	Угол поворота меток рисок (от 0 до 360)
`xticks`	Значения рисок на оси X
`yticks`	Значения рисок на оси Y
`xlim`	Границы по оси X (например, [0,10])
`ylim`	Границы по оси Y
`grid`	Отображать координатную сетку (по умолчанию включено)

Большинство методов построения графиков в pandas принимает необязательный параметр `ax` – объект подграфика matplotlib. Это позволяет гибко расположить подграфики в сетке. Более подробно я расскажу об этом в разделе о matplotlib API ниже.

Метод `plot` объекта DataFrame строит отдельные графики каждого столбца внутри одного подграфика и автоматически создает пояснительную надпись (см. рис. 8.14).

```
In [57]: df = DataFrame(np.random.randn(10, 4).cumsum(0),
   ....: columns=['A', 'B', 'C', 'D'],
   ....: index=np.arange(0, 100, 10))

In [58]: df.plot()
```

Дополнительные позиционные аргументы метода `plot` без изменения передаются соответствующей функции matplotlib, поэтому, внимательно изучив matplotlib API, вы сможете настраивать графики более точно.

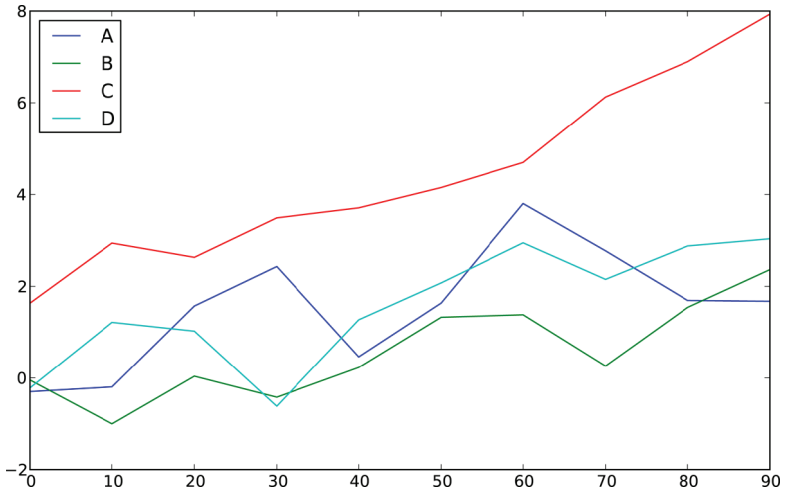


Рис. 8.14. Простой пример графика для объекта DataFrame

У объекта DataFrame есть ряд параметров, которые гибко описывают обработку столбцов, например, нужно ли строить их графики внутри одного и того же или разных подграфиков. Все они перечислены в табл. 8.4.

Таблица 8.4. Параметры метода DataFrame.plot

Аргумент	Описание
subplots	Рисовать график каждого столбца DataFrame в отдельном подграфике
`sharex`	Если `subplots=True`, то совместно использовать ось X, объединяя риски и границы
`sharey`	Если `subplots=True`, то совместно использовать ось Y
`figsize`	Размеры создаваемого рисунка в виде кортежа
title	Название графика в виде строки
legend	Помещать в подграфик пояснительную надпись (по умолчанию `True`)
`sort_columns`	Строить графики столбцов в алфавитном порядке; по умолчанию используется существующий порядок столбцов

О построении графиков временных рядов см. главу 10.

Столбчатые диаграммы

Чтобы построить не линейный график, а столбчатую диаграмму, достаточно передать параметр `kind='bar'` (если столбики должны быть вертикальными) или `kind='barh'` (если горизонтальными). В этом случае индекс Series или

DataFrame будет использоваться для нанесения рисок на оси X (`bar`) или Y (`barh`) (см. рис. 8.15):

```
In [59]: fig, axes = plt.subplots(2, 1)

In [60]: data = Series(np.random.rand(16), index=list('abcdefghijklmnop'))

In [61]: data.plot(kind='bar', ax=axes[0], color='k', alpha=0.7)
Out[61]: <matplotlib.axes.AxesSubplot at 0x4ee7750>

In [62]: data.plot(kind='barh', ax=axes[1], color='k', alpha=0.7)
```

Подробнее о функции `plt.subplots`, осях и рисунках в matplotlib, см. ниже в этой главе.

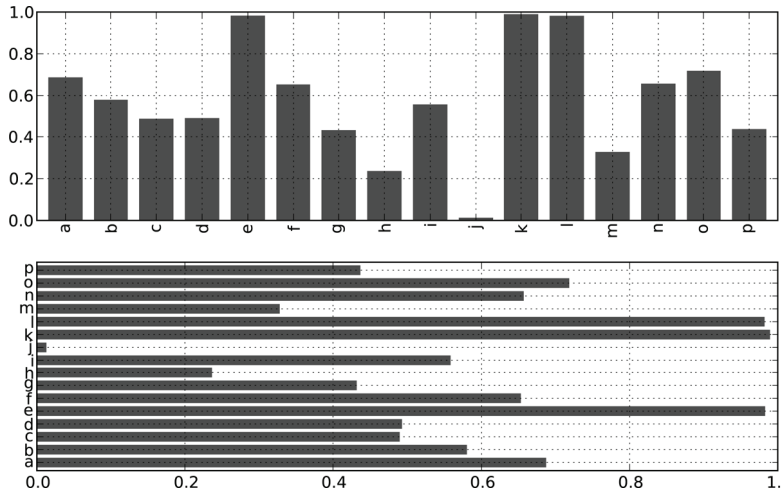


Рис. 8.15. Примеры горизонтальной и вертикальной столбчатых диаграмм

В случае DataFrame значения из каждой строки объединяются в группы столбиков, расположенные поодаль друг от друга. См. рис. 8.16.

```
In [63]: df = DataFrame(np.random.rand(6, 4),
   ....:                index=['one', 'two', 'three', 'four', 'five', 'six'],
   ....:                columns=pd.Index(['A', 'B', 'C', 'D'], name='Genus'))

In [64]: df
Out[64]:
Genus          A         B         C         D
one     0.301686  0.156333  0.371943  0.270731
two     0.750589  0.525587  0.689429  0.358974
three   0.381504  0.667707  0.473772  0.632528
four    0.942408  0.180186  0.708284  0.641783
five    0.840278  0.909589  0.010041  0.653207
```

```
six    0.062854  0.589813  0.811318  0.060217
In [65]: df.plot(kind='bar')
```

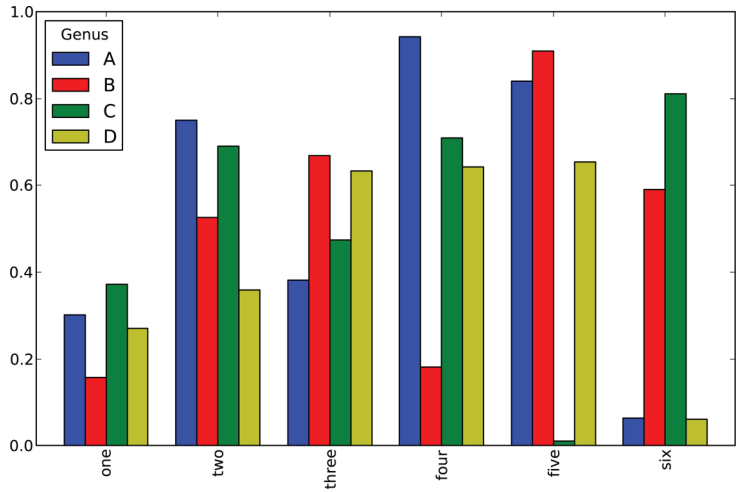


Рис. 8.16. Пример столбчатой диаграммы для DataFrame

Обратите внимание, что название столбцов DataFrame – «Genus» – используется в заголовке пояснительной надписи.

Для построения составной столбчатой диаграммы по объекту DataFrame нужно задать параметр `stacked=True`, тогда столбики, соответствующие значению в каждой строке, будут приставлены друг к другу (рис. 8.17):

```
In [67]: df.plot(kind='barh', stacked=True, alpha=0.5)
```

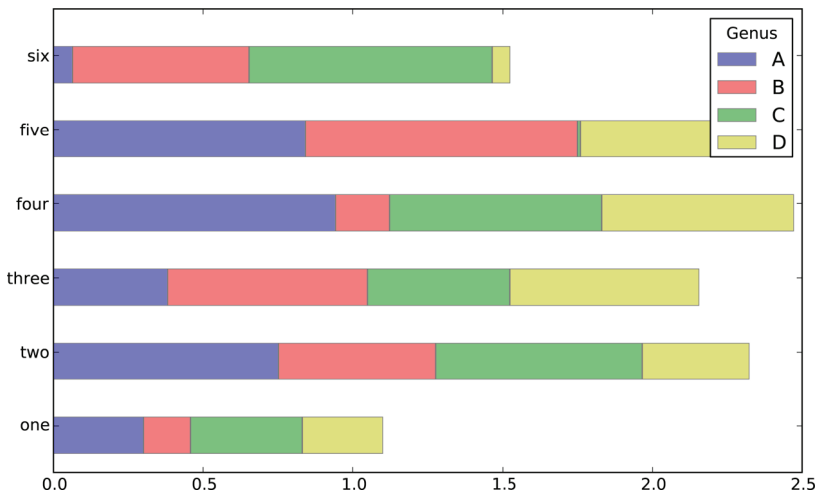


Рис. 8.17. Пример составной столбчатой диаграммы для DataFrame

Столбчатые диаграммы полезны для визуализации частоты значений в объекте Series с применением метода `value_counts`: `s.value_counts().plot(kind='bar')`. Мы использовали это в предыдущей главе.

Допустим, мы хотим построить составную столбчатую диаграмму, показывающую процентную долю данных, относящихся к каждому значению количества гостей в ресторане в каждый день. Я загружаю данные методом `read_csv` и выполняю кросс-табуляцию по дню и количеству гостей:

```
In [68]: tips = pd.read_csv('ch08/tips.csv')

In [69]: party_counts = pd.crosstab(tips.day, tips.size)

In [70]: party_counts
Out[70]:
size  1   2   3   4  5  6
day
Fri   1  16   1   1  0  0
Sat   2  53  18  13  1  0
Sun   0  39  15  18  3  1
Thur  1  48   4   5  1  3

# Группы, насчитывающие 1 и 6 гостей, редки
In [71]: party_counts = party_counts.ix[:, 2:5]
```

Затем нормирую значения, так чтобы сумма в каждой строке была равна 1 (во избежание проблем с целочисленным делением в версии Python 2.7 необходимо приводить к типу float), и строю график (рис. 8.18):

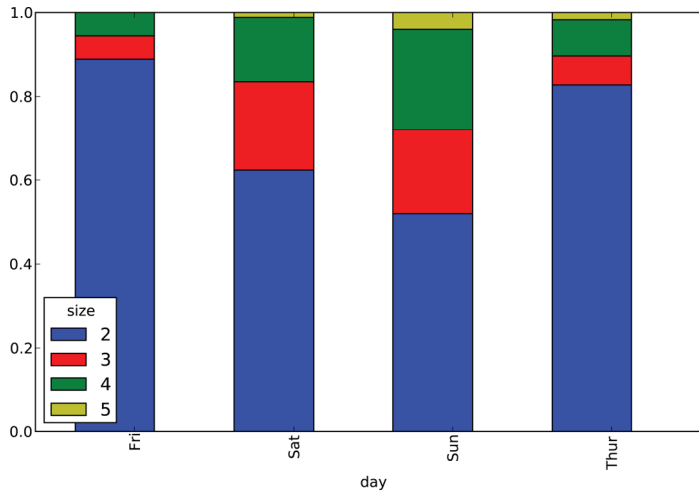


Рис. 8.18. Распределение по количеству гостей в группе за каждый день недели

```
# Нормирование на сумму 1
In [72]: party_pcts = party_counts.div(party_counts.sum(1).astype(float), axis=0)

In [73]: party_pcts
Out[73]:
size          2         3         4         5
day
Fri   0.888889  0.055556  0.055556  0.000000
Sat   0.623529  0.211765  0.152941  0.011765
Sun   0.520000  0.200000  0.240000  0.040000
Thur  0.827586  0.068966  0.086207  0.017241
In [74]: party_pcts.plot(kind='bar', stacked=True)
```

Как видим, в выходные количество гостей в одной группе увеличивается.

Гистограммы и графики плотности

Гистограмма, с которой все мы хорошо знакомы, – это разновидность столбчатой диаграммы, показывающая дискретизированное представление частоты. Результаты измерений распределяются по дискретным интервалам равной ширины, а на гистограмме отображается количество точек в каждом интервале. На примере приведенных выше данных о чаевых от гостей ресторана мы можем с помощью метода `hist` объекта Series построить гистограмму распределения процента чаевых от общей суммы счета (рис. 8.19):

```
In [76]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [77]: tips['tip_pct'].hist(bins=50)
```

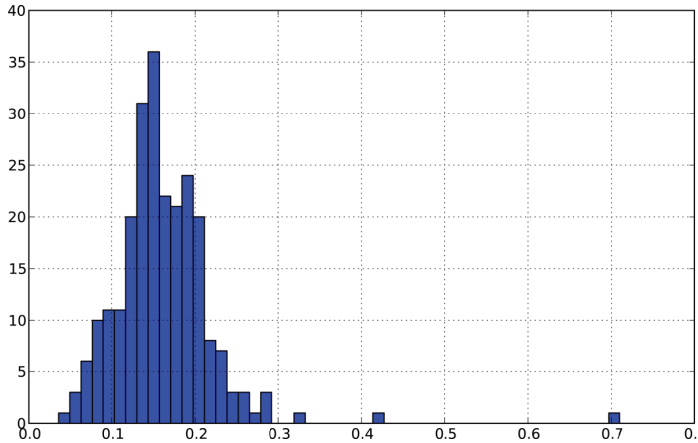


Рис. 8.19. Гистограмма процента чаевых

С гистограммой тесно связан *график плотности*, который строится на основе оценки непрерывного распределения вероятности по результатам измерений. Обычно стремятся аппроксимировать это распределение комбинацией *ядер*, т. е. более простых распределений, например нормального (гауссова). Поэтому графи-

ки плотности еще называют графиками ядерной оценки плотности (KDE – kernel density estimate). Функция `plot` с параметром `kind='kde'` строит график плотности, применяя стандартный метод комбинирования нормальных распределений (рис. 8.20):

```
In [79]: tips['tip_pct'].plot(kind='kde')
```

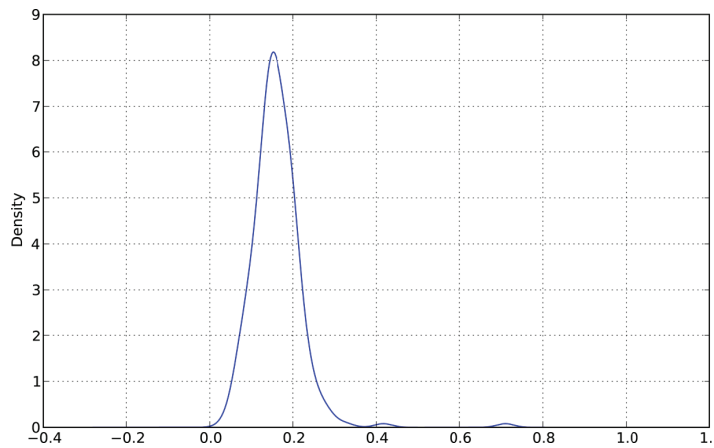


Рис. 8.20. График плотности процента чаевых

Эти два вида графиков часто рисуются вместе: гистограмма в нормированном виде (показывающая дискретизированную плотность), а поверх нее – график ядерной оценки плотности. В качестве примера рассмотрим бимодальное распределение, образованное выборками из двух стандартных нормальных распределений (рис. 8.21):

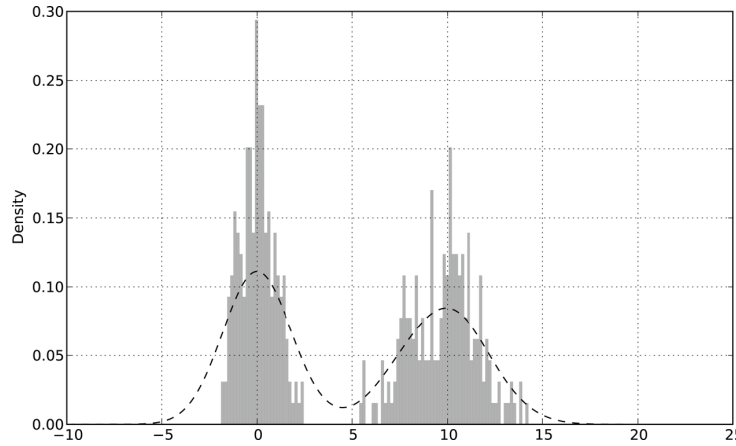


Рис. 8.21. Нормированная гистограмма двух нормальных распределений в сочетании с оценкой плотности

```
In [81]: comp1 = np.random.normal(0, 1, size=200) # N(0, 1)

In [82]: comp2 = np.random.normal(10, 2, size=200) # N(10, 4)

In [83]: values = Series(np.concatenate([comp1, comp2]))

In [84]: values.hist(bins=100, alpha=0.3, color='k', normed=True)
Out[84]: <matplotlib.axes.AxesSubplot at 0x5cd2350>

In [85]: values.plot(kind='kde', style='k--')
```

Диаграммы рассеяния

Диаграмма рассеяния – полезный способ исследования соотношения между двумя одномерными рядами данных. В пакете matplotlib есть метод `scatter`, с помощью которого строятся все такого рода графики. Для демонстрации я загрузил набор данных `macrodata` из проекта statsmodels, выбрал несколько переменных и вычислил логарифмические разности:

```
In [86]: macro = pd.read_csv('ch08/macrodata.csv')

In [87]: data = macro[['cpi', 'm1', 'tbilrate', 'unemp']]

In [88]: trans_data = np.log(data).diff().dropna()

In [89]: trans_data[-5:]
Out[89]:
          cpi        m1   tbilrate     unemp
198 -0.007904  0.045361  -0.396881  0.105361
199 -0.021979  0.066753  -2.277267  0.139762
200  0.002340  0.010286   0.606136  0.160343
201  0.008419  0.037461  -0.200671  0.127339
202  0.008894  0.012202  -0.405465  0.042560
```

Метод `plt.scatter` позволяет без труда построить простую диаграмму рассеяния (рис. 8.22):

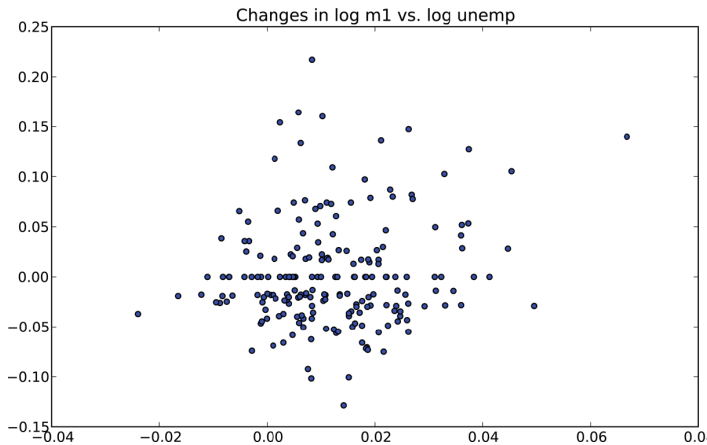


Рис. 8.22. Простая диаграмма рассеяния

```
In [91]: plt.scatter(trans_data['m1'], trans_data['unemp'])
Out[91]: <matplotlib.collections.PathCollection at 0x43c31d0>

In [92]: plt.title('Changes in log %s vs. log %s' % ('m1', 'unemp'))
```

В разведочном анализе данных полезно видеть все диаграммы рассеяния для группы переменных; это называется *матрицей диаграмм рассеяния*. Построение такого графика с нуля – довольно утомительное занятие, поэтому в pandas имеется функция `scatter_matrix` для построения матрицы на основе объекта DataFrame. Она поддерживает также размещение гистограмм или графиков плотности каждой переменной вдоль диагонали. Результирующий график показан на рис. 8.23:

```
In [93]: scatter_matrix(trans_data, diagonal='kde', color='k', alpha=0.3)
```

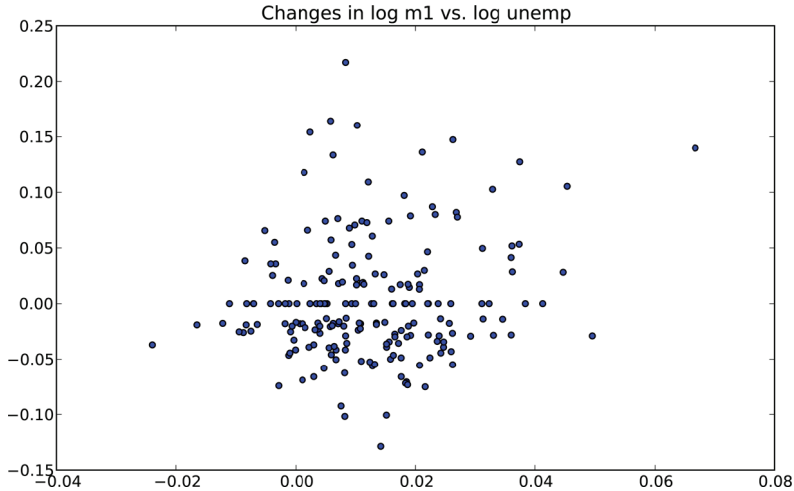


Рис. 8.23. Матрица диаграмма рассеяния для набора данных macrodata из проекта statsmodels

Нанесение данных на карту: визуализация данных о землетрясении на Гаити

Ushahidi – некоммерческая компания, предлагающая программные средства для привлечения широких масс Интернет-общественности к сбору данных о природных катастрофах и геополитических событиях путем отправки текстовых сообщений. Многие данные, полученные таким образом, затем публикуются на сайте компании по адресу *http://community.ushahidi.com* для анализа и визуализации. Я загрузил данные о землетрясении 2010 года на Гаити и его последствиях и покажу, как подготовил их для анализа и визуализации с помощью pandas и других

рассмотренных выше инструментов. Скачав CSV-файл с указанного сайта, мы можем загрузить его в объект DataFrame методом `read_csv`:

```
In [94]: data = pd.read_csv('ch08/Haiti.csv')

In [95]: data
Out[95]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3593 entries, 0 to 3592
Data columns:
Serial           3593  non-null values
INCIDENT TITLE   3593  non-null values
INCIDENT DATE    3593  non-null values
LOCATION         3593  non-null values
DESCRIPTION      3593  non-null values
CATEGORY         3587  non-null values
LATITUDE         3593  non-null values
LONGITUDE        3593  non-null values
APPROVED         3593  non-null values
VERIFIED         3593  non-null values
dtypes: float64(2), int64(1), object(7)
```

Теперь нетрудно поработать с этими данными и посмотреть, что из них можно извлечь. Каждая строка представляет собой отправленный с чьего-то мобильного телефона отчет с описанием чрезвычайной ситуации или еще какой-то проблемы. С каждым отчетом ассоциирована временная метка и географическая привязка в виде широты и долготы:

```
In [96]: data[['INCIDENT DATE', 'LATITUDE', 'LONGITUDE']][:10]
Out[96]:
INCIDENT DATE LATITUDE LONGITUDE
0  05/07/2010 17:26  18.233333   -72.533333
1  28/06/2010 23:06  50.226029     5.729886
2  24/06/2010 16:21  22.278381   114.174287
3  20/06/2010 21:59  44.407062     8.933989
4  18/05/2010 16:26  18.571084   -72.334671
5  26/04/2010 13:14  18.593707   -72.310079
6  26/04/2010 14:19  18.482800   -73.638800
7  26/04/2010 14:27  18.415000   -73.195000
8  15/03/2010 10:58  18.517443   -72.236841
9  15/03/2010 11:00  18.547790   -72.410010
```

В поле `CATEGORY` находится список разделенных запятыми кодов, описывающих тип сообщения:

```
In [97]: data['CATEGORY'][:6]
Out[97]:
0         1. Urgences | Emergency, 3. Public Health,
1  1. Urgences | Emergency, 2. Urgences logistiques
2  2. Urgences logistiques | Vital Lines, 8. Autre |
3                           1. Urgences | Emergency,
4                           1. Urgences | Emergency,
5                     5e. Communication lines down,
Name: CATEGORY
```

Из приведенной выше сводки видно, что некоторые категории отсутствуют, поэтому такие данные лучше отбросить. Кроме того, метод `describe` показывает наличие аномальных координат:

```
In [98]: data.describe()
Out[98]:
Serial LATITUDE LONGITUDE
count  3593.000000  3593.000000  3593.000000
mean   2080.277484    18.611495   -72.322680
std    1171.100360     0.738572     3.650776
min       4.000000    18.041313   -74.452757
25%    1074.000000    18.524070   -72.417500
50%    2163.000000    18.539269   -72.335000
75%    3088.000000    18.561820   -72.293570
max    4052.000000    50.226029   114.174287
```

Вычистить заведомо некорректные координаты и удалить отсутствующие категории теперь совсем несложно:

```
In [99]: data = data[(data.LATITUDE > 18) & (data.LATITUDE < 20) &
   ....:              (data.LONGITUDE > -75) & (data.LONGITUDE < -70)
   ....:              & data.CATEGORY.notnull()]
```

Теперь можно было бы провести анализ или визуализацию этих данных по категориям, однако в каждом поле категории может находиться несколько категорий. Кроме того, каждая категория представлена кодом и английским, а, возможно, также и французским названием. Поэтому данные следует переформатировать, приведя их к более подходящему для анализа виду. Для начала я написал две функции, которые получают список всех категорий и выделяют из каждой категории код и английское название:

```
def to_cat_list(catstr):
  stripped = (x.strip() for x in catstr.split(','))
  return [x for x in stripped if x]

def get_all_categories(cat_series):
  cat_sets = (set(to_cat_list(x)) for x in cat_series)
  return sorted(set.union(*cat_sets))

def get_english(cat):
  code, names = cat.split('.')
  if '|' in names:
     names = names.split(' | ')[1]
  return code, names.strip()
```

Можем протестировать, что функция `get_english` действительно возвращает то, что мы ожидаем:

```
In [101]: get_english('2. Urgences logistiques | Vital Lines')
Out[101]: ('2', 'Vital Lines')
```

Затем я строю словарь, отображающий код на название, поскольку для анализа мы будем использовать коды. Этот словарь понадобится позже для оформления

графиков (обратите внимание на использование генераторного выражения вместо спискового включения):

```
In [102]: all_cats = get_all_categories(data.CATEGORY)

# Генераторное выражение
In [103]: english_mapping = dict(get_english(x) for x in all_cats)

In [104]: english_mapping['2a']
Out[104]: 'Food Shortage'

In [105]: english_mapping['6c']
Out[105]: 'Earthquake and aftershocks'
```

Есть много способов пополнить этот набор данных, чтобы упростить выборку записей по категории. Например, можно добавить индикаторные (фиктивные) столбцы, по одному для каждой категории. Для этого мы сначала выделим уникальные коды категорий и построим объект DataFrame из одних нулей, в которых эти коды будут столбцами, а индекс будет таким же, как у `data`:

```
def get_code(seq):
  return [x.split('.')[0] for x in seq if x]

all_codes = get_code(all_cats)
code_index = pd.Index(np.unique(all_codes))
dummy_frame = DataFrame(np.zeros((len(data), len(code_index))),
                        index=data.index, columns=code_index)
```

Если не возникнет ошибок, то `dummy_frame` должен выглядеть так:

```
In [107]: dummy_frame.ix[:, :6]

Out[107]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3569 entries, 0 to 3592
Data columns:
1    3569  non-null values
1a   3569  non-null values
1b   3569  non-null values
1c   3569  non-null values
1d   3569  non-null values
2    3569  non-null values
dtypes: float64(6)
```

Как вы помните, осталось сделать нужные элементы в каждой строке равными 1, после чего соединить это объект с `data`:

```
for row, cat in zip(data.index, data.CATEGORY):
  codes = get_code(to_cat_list(cat))
  dummy_frame.ix[row, codes] = 1

data = data.join(dummy_frame.add_prefix('category_'))
```

Теперь в data появились новые столбцы:

```
In [109]: data.ix[:, 10:15]
Out[109]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3569 entries, 0 to 3592
Data columns:
category_1    3569  non-null values
category_1a   3569  non-null values
category_1b   3569  non-null values
category_1c   3569  non-null values
category_1d   3569  non-null values
dtypes: float64(5)
```

Пора переходить к построению графиков! Поскольку это географические данные, мы хотели бы нанести их на карту Гаити — отдельно для каждой категории. Пакет basemap (*http://matplotlib.github.com/basemap*), дополнение к matplotlib, позволяет наносить двумерные данные на карты из программ на Python. Пакет поддерживает несколько географических проекций и средства для преобразования широты и долготы в двумерный график matplotlib. Двигаясь методом проб и ошибок и используя для экспериментов приведенные выше данные, я в конце концов написал функцию, которая рисует простую черно-белую карту Гаити:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt

def basic_haiti_map(ax=None, lllat=17.25, urlat=20.25,
                    lllon=-75, urlon=-71):
  # создать экземпляр Basemap в полярной стереографической проекции
  m = Basemap(ax=ax, projection='stere',
              lon_0=(urlon + lllon) / 2,
              lat_0=(urlat + lllat) / 2,
              llcrnrlat=lllat, urcrnrlat=urlat,
              llcrnrlon=lllon, urcrnrlon=urlon,
              resolution='f')
 # нарисовать береговую линию, границы штатов и страны, край карты
 m.drawcoastlines()
 m.drawstates()
 m.drawcountries()
 return m
```

Идея в том, что возвращенный объект Basemap знает, как нанести точку с данными координатами на холст. Приведенный ниже код строит графики данных наблюдений для различных категорий в отчетах. Для каждой категории я отбираю из набора данных только координаты, аннотированные данной категорией, создаю Basemap в соответствующем подграфике, преобразую координаты и наношу точки с помощью метода plot объекта Basemap:

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(12, 10))
fig.subplots_adjust(hspace=0.05, wspace=0.05)

to_plot = ['2a', '1', '3c', '7a']
```

```
lllat=17.25; urlat=20.25; lllon=-75; urlon=-71

for code, ax in zip(to_plot, axes.flat):
  m = basic_haiti_map(ax, lllat=lllat, urlat=urlat,
                      lllon=lllon, urlon=urlon)

  cat_data = data[data['category_%s' % code] == 1]

  # вычислить координаты проекции на карте
  x, y = m(cat_data.LONGITUDE, cat_data.LATITUDE)

  m.plot(x, y, 'k.', alpha=0.5)
  ax.set_title('%s: %s' % (code, english_mapping[code]))
```

Получившийся рисунок показан на рис. 8.24.

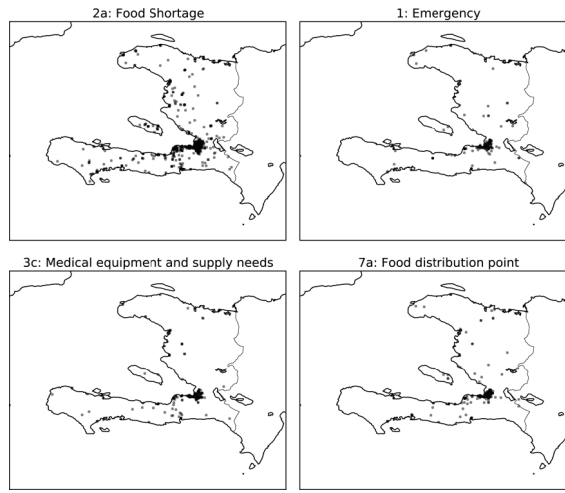


Рис. 8.24. Данные о землетрясении на Гаити по четырем категориям

Из графиков видно, что большая часть данных концентрируется вокруг самого густонаселенного города, Порт-о-Пренс. Пакет `basemap` позволяет наносить на карту дополнительные слои, данные для которых берутся из так называемых *shape-файлов*. Сначала я загрузил shape-файл с данными о дорогах в Порт-о-Пренсе (*http://cegrp.cga.harvard.edu/haiti/?q=resources_data*). Объект `Basemap` любезно предоставляет метод `readshapefile`, поэтому распаковав архив, я просто добавил в код следующие строчки:

```
shapefile_path = 'ch08/PortAuPrince_Roads/PortAuPrince_Roads'
m.readshapefile(shapefile_path, 'roads')
```

Еще немного поковырявшись с границами широты и долготы, я методом проб и ошибок все-таки построил графики для категории «Food shortage» (Недостаток продуктов питания); он показан на рис. 8.25.

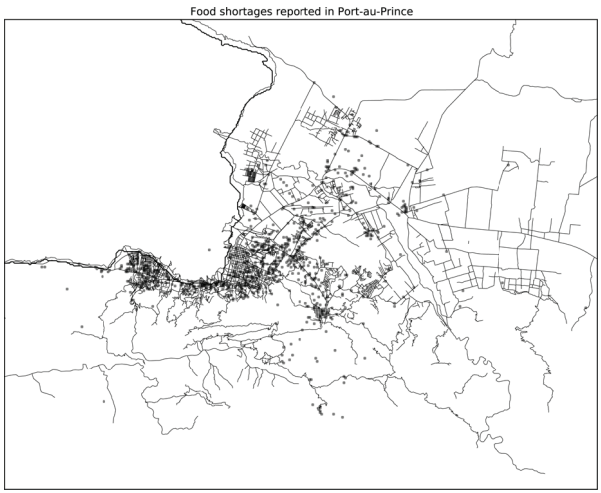


Рис. 8.25. Отчеты о недостатке продуктов питания в Порт-о-Пренсе во время землетрясения на Гаити

Инструментальная экосистема визуализации для Python

Как обычно бывает в проектах с открытым исходным кодом, в средствах создания графики для Python нехватки не ощущается (их слишком много, чтобы все перечислить). Помимо открытого кода, существуют также многочисленные коммерческие библиотеки с интерфейсом к Python.

В этой главе и в книге в целом я использую, в основном пакет matplotlib, потому что это наиболее популярное средство построения графиков в Python. Но хотя matplotlib является важной частью научной экосистемы Python, у него обнаруживается немало недостатков, когда дело доходит до создания и отображения статистических графиков. Пользователям MATLAB пакет matplotlib покажется знакомым, тогда как пользователи R (особенно работающие с великолепными пакетами `ggplot2` и `trellis`), наверное, испытают разочарование (по крайней мере, на момент написания этой книги). С помощью matplotlib можно создать отличные графики для отображения в веб, но для этого обычно приходится прикладывать немало усилий, потому что библиотека проектировалась для полиграфических изданий. Но если не обращать внимания на эстетические нюансы, то для большинства задач этого достаточно. В pandas я, наряду с другими разработчиками, стремился предложить пользователям удобный интерфейс, позволяющий упростить построение большинства графиков, применяемых в анализе данных.

Но существуют и активно используются и другие средства визуализации. Ниже я перечислю некоторых из них и призываю вас самостоятельно исследовать экосистему.

Chaco

Пакет Chaco (*http://code.enthought.com/chaco/*), разработанный компанией Enthought, представляет собой инструментарий для построения графиков. Он годится как для рисования статических графиков, так и для интерактивной визуализации. Особенно хорош он, когда нужно выразить сложные визуализации, характеризуемые наличием взаимозависимостей между данными. По сравнению с matplotlib Chaco гораздо лучше поддерживает взаимодействие с элементами графика, а отрисовка производится очень быстро, так что этот пакет будет хорошим выбором для построения интерактивных приложений с графическим интерфейсом.

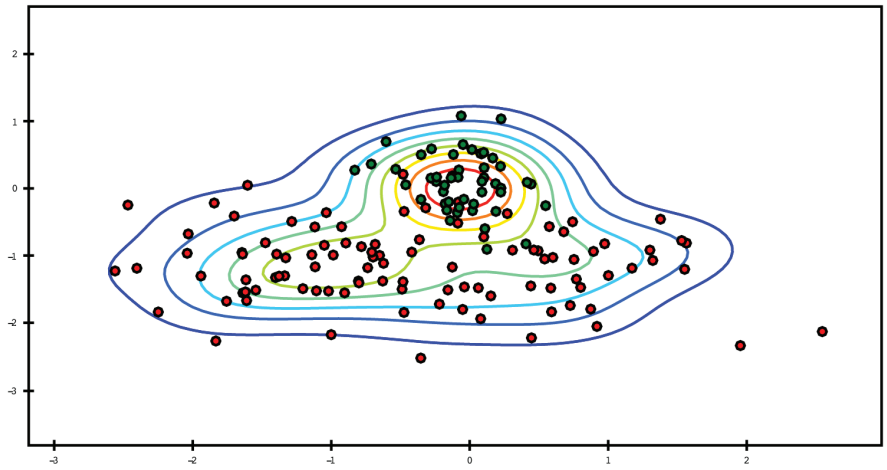


Рис. 8.26. Пример графика, построенного Chaco

mayavi

Проект mayavi, разработанный Прабху Рамачандраном (Prabhu Ramachandran) Гаэлем Вароко (Gaël Varoquaux) и другими, – это библиотека трехмерной графики, построенная поверх написанной на C++ библиотеки VTK с открытым исходным кодом. mayavi, как и matplotlib, интегрирована с IPython, поэтому с ней легко работать интерактивно. Графики можно панорамировать, поворачивать и масштабировать с помощью мыши и клавиатуры. Я использовал mayavi для создания одной иллюстрации укладывания в главе 12. И хотя я не привожу здесь свой код работы с mayavi, в сети достаточно документации и примеров. Я полагаю, что во многих случаях это неплохая альтернатива таким технологиям, как WebGL, хотя интерактивными графиками труднее поделиться.

Прочие пакеты

Разумеется, для Python хватает и других библиотек и приложений для визуализации: PyQwt, Veusz, gnuplot-py, biggles и т. д. Я видел, как удачное применение PyQwt в графических приложениях, написанных с помощью каркаса Qt и надстройки PyQt над ним. Многие библиотеки продолжают активно развиваться (некоторые являются частью гораздо более крупных приложений), но в последние годы наблюдается общая тенденция к дрейфу в сторону веб-технологий и отхода от графики для настольных компьютеров. Я еще скажу об этом несколько слов в следующем разделе.

Будущее средств визуализации

Похоже, будущее за веб-технологиями (на основе JavaScript), и от этого никуда не деться. Без сомнения, за прошедшие годы вы встречались с самыми разными видами статических и интерактивных визуализаций, написанных на Flash или JavaScript. Постоянно появляются все новые и новые инструменты (например, d3.js и многочисленные отпочковавшиеся от него проекты). Напротив, разработка средств визуализации на платформах, отличных от веб, в последние годы резко замедлилась. Это относится как к Python, так и к другим средам для анализа данных и статистических расчетов, например R.

Поэтому задача разработчика видится в том, чтобы организовать более тесную интеграцию между средствами анализа и подготовки данных, в частности pandas, и веб-браузером. Я надеюсь, что это также станет областью плодотворного сотрудничества между пишущими на Python и на других языках.

ГЛАВА 9. Агрегирование данных и групповые операции

Разбиение набора данных на группы и применение некоторой функции к каждой группе, будь то в целях агрегирования или преобразования, зачастую является одной из важнейших операций анализа данных. После загрузки, слияния и подготовки набора данных обычно вычисляют статистику по группам или, возможно, *сводные таблицы* для построения отчета или визуализации. В библиотеке pandas имеется гибкий и быстрый механизм `groupby`, который позволяет формировать продольные и поперечные срезы, а также агрегировать наборы данных естественным образом.

Одна из причин популярности реляционных баз данных и языка SQL (структурированного языка запросов) – простота соединения, фильтрации, преобразования и агрегирования данных. Однако в том, что касается групповых операций, языки запросов типа SQL несколько ограничены. Как мы увидим, выразительность и мощь языка Python и библиотеки pandas позволяют выполнять гораздо более сложные групповые операции с помощью функций, принимающих произвольный объект pandas или массив NumPy. В этой главе мы изучим следующие возможности:

- разделение объекта pandas на части по одному или нескольким ключам (в виде функций, массивов или имен столбцов объекта DataFrame);
- вычисление групповых статистик, например: общее количество, среднее, стандартное отклонение или определенная пользователем функция;
- применение переменного множества функций к каждому столбцу DataFrame;
- применение внутригрупповых преобразований или иных манипуляций, например: нормировка, линейная регрессия, ранжирование или выборка подмножества;
- вычисление сводных таблиц и таблиц сопряженность;
- квантильный анализ и другие вида анализа групп.

Частный случай механизма `groupby`, агрегирование временных рядов, в этой книге называется *передискретизацией* и рассматривается отдельно в главе 10.

Механизм GroupBy

Хэдли Уикхэм (Hadley Wickham), автор многих популярных пакетов на языке программирования R, предложил для групповых операций термин *разделение-применение-объединение*, который, как мне кажется, удачно описывает процесс. На первом этапе данные, хранящиеся в объекте pandas, будь то Series, DataFrame или что-то еще, *разделяются* на группы по одному или нескольким указанными вами *ключам*. Разделение производится вдоль одной оси объекта. Например, DataFrame можно группировать по строкам (`axis=0`) или по столбцам (`axis=1`). Затем к каждой группе *применяется* некоторая функция, которая порождает новое значение. Наконец, результаты применения всех функций *объединяются* в результирующий объект. Форма результирующего объекта обычно зависит от того, что именно проделывается с данными. На рис. 9.1 показан схематический пример простого группового агрегирования.

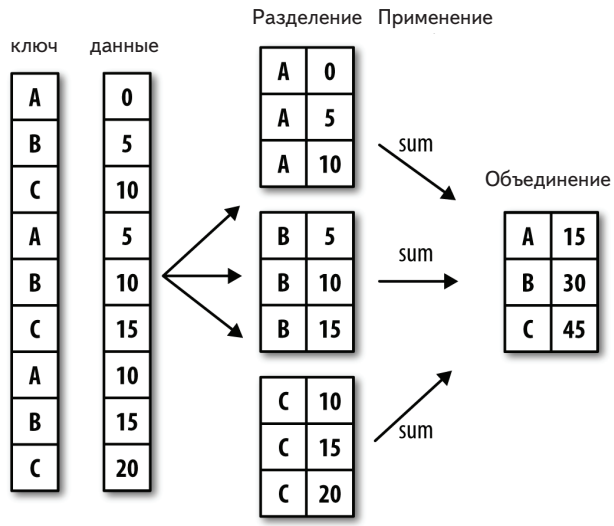


Рис. 9.1. Пример группового агрегирования

Ключи группировки могут задаваться по-разному и необязательно должны быть одного типа:

- список или массив значений той же длины, что ось, по которой производится группировка;
- значение, определяющее имя столбца объекта DataFrame;
- словарь или объект Series, определяющий соответствие между значениями на оси группировки и именами групп;
- функция, которой передается индекс оси или отдельные метки из этого индекса.

Отметим, что последние три метода – просто различные способы порождения массива значений, используемого далее для разделения объекта на группы. Не пу-

гайтесь, если это кажется слишком абстрактным. В этой главе будут приведены многочисленные примеры каждого метода. Для начала рассмотрим очень простой табличный набор данных, представленный в виде объекта DataFrame:

```
In [13]: df = DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
   ....:                 'key2' : ['one', 'two', 'one', 'two', 'one'],
   ....:                 'data1' : np.random.randn(5),
   ....:                 'data2' : np.random.randn(5)})

In [14]: df
Out[14]:
       data1     data2  key1  key2
0  -0.204708  1.393406     a   one
1   0.478943  0.092908     a   two
2  -0.519439  0.281746     b   one
3  -0.555730  0.769023     b   two
4   1.965781  1.246435     a   one
```

Пусть требуется вычислить среднее по столбцу `data1`, используя метки групп в столбце `key1`. Сделать это можно несколькими способами. Первый – взять столбец `data1` и вызвать метод `groupby`, передав ему столбец (объект Series) `key1`:

```
In [15]: grouped = df['data1'].groupby(df['key1'])

In [16]: grouped
Out[16]: <pandas.core.groupby.SeriesGroupBy at 0x2d78b10>
```

Переменная `grouped` – это объект `GroupBy`. Пока что он не вычислил ничего, кроме промежуточных данных о групповом ключе `df['key1']`. Идея в том, что этот объект хранит всю информацию, необходимую для последующего применения некоторой операции к каждой группе. Например, чтобы вычислить средние по группам, мы можем вызвать метод `mean` объекта GroupBy:

```
In [17]: grouped.mean()
Out[17]:
         key1
a    0.746672
b   -0.537585
```

Позже я подробнее объясню, что происходит при вызове `.mean()`. Важно, что данные (объект Series) агрегированы по групповому ключу, и в результате создан новый объект Series, индексированный уникальными значениями в столбце `key1`. Получившийся индекс назван `'key1'`, потому что так назывался столбец `df['key1']` объекта DataFrame.

Если бы мы передали несколько массивов в виде списка, то получили бы другой результат:

```
In [18]: means = df['data1'].groupby([df['key1'], df['key2']]).mean()

In [19]: means
Out[19]:
```

```
   key1       key2
a   one   0.880536
    two   0.478943
b   one  -0.519439
    two  -0.555730
```

В данном случае данные сгруппированы по двум ключам, а у результирующего объекта Series имеется иерархический индекс, который состоит из уникальных пар значений ключей, встретившихся в исходных данных:

```
In [20]: means.unstack()
Out[20]:
key2     one        two
key1
a   0.880536   0.478943
b  -0.519439  -0.555730
```

В этих примерах групповыми ключами были объекты Series, но можно было бы использовать и произвольные массивы правильной длины:

```
In [21]: states = np.array(['Ohio', 'California', 'California', 'Ohio', 'Ohio'])

In [22]: years = np.array([2005, 2005, 2006, 2005, 2006])

In [23]: df['data1'].groupby([states, years]).mean()
Out[23]:
California  2005   0.478943
            2006  -0.519439
Ohio        2005  -0.380219
            2006   1.965781
```

Часто информация о группировке находится в том же объекте DataFrame, что и группируемые данные. В таком случае в качестве групповых ключей можно передать имена столбцов (неважно, что они содержат: строки, числа или другие объекты Python):

```
In [24]: df.groupby('key1').mean()
Out[24]:
       data1     data2
key1
a   0.746672  0.910916
b  -0.537585  0.525384

In [25]: df.groupby(['key1', 'key2']).mean()
Out[25]:
          data1     data2
key1 key2
a  one  0.880536  1.319920
   two  0.478943  0.092908
b  one -0.519439  0.281746
   two -0.555730  0.769023
```

Вероятно, вы обратили внимание, что в первом случае – `df.groupby('key1').mean()` – результат не содержал столбца `key2`. Поскольку `df['key2']` содержит

нечисловые данные, говорят, что это *посторонний столбец*, и в результат не включают. По умолчанию агрегируются все числовые столбцы, хотя можно выбрать и некоторое их подмножество, как мы вскоре увидим.

Вне зависимости от цели использования `groupby` у объекта GroupBy есть полезный в любом случае метод `size`, который возвращает объект Series, содержащий размеры групп:

```
In [26]: df.groupby(['key1', 'key2']).size()
Out[26]:
key1 key2
a    one     2
     two     1
b    one     1
     two     1
```

На момент написания книги данные, соответствующие отсутствующим значениям группового ключа, исключались из результата. Возможно (и даже очень вероятно), что к моменту выхода книги из печати появится параметр, разрешающий включать в результат группу `NA`.

Обход групп

Объект GroupBy поддерживает итерирование, в результате которого генерируется последовательность 2-кортежей, содержащих имя группы и блок данных. Рассмотрим следующий небольшой набор данных:

```
In [27]: for name, group in df.groupby('key1'):
   ....:     print name
   ....:     print group
   ....:
a
       data1     data2  key1  key2
0  -0.204708  1.393406     a   one
1   0.478943  0.092908     a   two
4   1.965781  1.246435     a   one
b
       data1     data2  key1  key2
2  -0.519439  0.281746     b   one
3  -0.555730  0.769023     b   two
```

В случае нескольких ключей первым элементом кортежа будет кортеж, содержащий значения ключей:

```
In [28]: for (k1, k2), group in df.groupby(['key1', 'key2']):
   ....:     print k1, k2
   ....:     print group
   ....:
a one
       data1     data2  key1  key2
0  -0.204708  1.393406     a   one
4   1.965781  1.246435     a   one
a two
```

```
        data1     data2  key1  key2
1   0.478943  0.092908     a   two
b one
        data1     data2  key1  key2
2  -0.519439  0.281746     b   one
b two
        data1     data2  key1  key2
3  -0.555730  0.769023     b   two
```

Разумеется, только вам решать, что делать с блоками данных. Возможно, пригодится следующий однострочный код, который строит словарь блоков:

```
In [29]: pieces = dict(list(df.groupby('key1')))

In [30]: pieces['b']
Out[30]:
       data1     data2  key1  key2
2  -0.519439  0.281746     b   one
3  -0.555730  0.769023     b   two
```

По умолчанию метод `groupby` группирует по оси `axis=0`, но можно задать любую другую ось. Например, в нашем примере столбцы объекта `df` можно было бы сгруппировать по `dtype`:

```
In [31]: df.dtypes
Out[31]:
data1   float64
data2   float64
key1    object
key2    object

In [32]: grouped = df.groupby(df.dtypes, axis=1)

In [33]: dict(list(grouped))
Out[33]:
{dtype('float64'):        data1        data2
0 -0.204708  1.393406
1  0.478943  0.092908
2 -0.519439  0.281746
3 -0.555730  0.769023
4  1.965781  1.246435,
 dtype('object'): key1 key2
0   a  one
1   a  two
2   b  one
3   b  two
4   a  one}
```

Выборка столбца или подмножества столбцов

Доступ по индексу к объекту GroupBy, полученному группировкой объекта DataFrame путем задания имени столбца или массива имен столбцов, имеет тот же эффект, что *выборка этих столбцов* для агрегирования. Это означает, что

```
df.groupby('key1')['data1']
df.groupby('key1')[['data2']]
```

– в точности то же самое, что:

```
df['data1'].groupby(df['key1'])
df[['data2']].groupby(df['key1'])
```

Большие наборы данных обычно желательно агрегировать лишь по немногим столбцам. Например, чтобы в приведенном выше примере вычислить среднее только по столбцу `data2` и получить результат в виде DataFrame, можно было бы написать:

```
In [34]: df.groupby(['key1', 'key2'])[['data2']].mean()
Out[34]:
data2
   key1      key2
a   one  1.319920
    two  0.092908
b   one  0.281746
    two  0.769023
```

В результате этой операции доступа по индексу возвращается сгруппированный DataFrame, если передан список или массив, или сгруппированный Series, если передано только одно имя столбца:

```
In [35]: s_grouped = df.groupby(['key1', 'key2'])['data2']

In [36]: s_grouped
Out[36]: <pandas.core.groupby.SeriesGroupBy at 0x2e215d0>

In [37]: s_grouped.mean()
Out[37]:
   key1      key2
a   one  1.319920
    two  0.092908
b   one  0.281746
    two  0.769023
```

Группировка с помощью словарей и объектов Series

Информацию о группировке можно передавать не только в виде массива. Рассмотрим еще один объект DataFrame:

```
In [38]: people = DataFrame(np.random.randn(5, 5),
   ....:                    columns=['a', 'b', 'c', 'd', 'e'],
   ....:                    index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis'])

In [39]: people.ix[2:3, ['b', 'c']] = np.nan # Add a few NA values
In [40]: people
Out[40]:
```

```
              a         b         c         d         e
Joe     1.007189 -1.296221  0.274992  0.228913  1.352917
Steve   0.886429 -2.001637 -0.371843  1.669025 -0.438570
Wes    -0.539741       NaN       NaN -1.021228 -0.577087
Jim     0.124121  0.302614  0.523772  0.000940  1.343810
Travis -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Теперь предположим, что имеется соответствие между столбцами и группами, и нужно просуммировать столбцы для каждой группы:

```
In [41]: mapping = {'a': 'red', 'b': 'red', 'c': 'blue',
   ....:            'd': 'blue', 'e': 'red', 'f' : 'orange'}
```

Из этого словаря нетрудно построить массив и передать его `groupby`, но можно вместо этого передать и сам словарь:

```
In [42]: by_column = people.groupby(mapping, axis=1)

In [43]: by_column.sum()
Out[43]:
             blue       red
Joe      0.503905  1.063885
Steve    1.297183 -1.553778
Wes     -1.021228 -1.116829
Jim      0.524712  1.770545
Travis  -4.230992 -2.405455
```

То же самое относится и к объекту Series, который можно рассматривать как отображение фиксированного размера. Когда в рассмотренных выше примерах я применял объект Series для задания групповых ключей, pandas на самом деле проверяла, что индекс Series выровнен с осью, по которой производится группировка:

```
In [44]: map_series = Series(mapping)

In [45]: map_series
Out[45]:
a     red
b     red
c    blue
d    blue
e     red
f  orange

In [46]: people.groupby(map_series, axis=1).count()
Out[46]:
        blue  red
Joe        2    3
Steve      2    3
Wes        1    2
Jim        2    3
Travis     2    3
```

Группировка с помощью функций

Использование функции Python – более абстрактный способ определения соответствия групп по сравнению со словарями или объектами Series. Функция, переданная в качестве группового ключа, будет вызвана по одному разу для каждого значения в индексе, а возвращенные ей значения станут именами групп. Конкретно, рассмотрим пример объекта DataFrame из предыдущего раздела, где значениями индекса являются имена людей. Пусть требуется сгруппировать по длине имени; можно было бы вычислить массив длин строк, но лучше вместо этого просто передать функцию `len`:

```
In [47]: people.groupby(len).sum()
Out[47]:
          a         b         c         d         e
3  0.591569 -0.993608  0.798764 -0.791374  2.119639
5  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Использование вперемежку функций, массивов, словарей и объектов Series вполне допустимо, потому что внутри все преобразуется в массивы:

```
In [48]: key_list = ['one', 'one', 'one', 'two', 'two']

In [49]: people.groupby([len, key_list]).min()
Out[49]:
              a         b         c         d         e
3 one -0.539741 -1.296221  0.274992 -1.021228 -0.577087
  two  0.124121  0.302614  0.523772  0.000940  1.343810
5 one  0.886429 -2.001637 -0.371843  1.669025 -0.438570
6 two -0.713544 -0.831154 -2.370232 -1.860761 -0.860757
```

Группировка по уровням индекса

Наконец, иерархически индексированные наборы данных можно агрегировать по одному из уровней индекса оси. Для этого нужно передать номер или имя уровня в именованном параметре `level`:

```
In [50]: columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
   ....:                                      [1, 3, 5, 1, 3]], names=['cty', 'tenor'])

In [51]: hier_df = DataFrame(np.random.randn(4, 5), columns=columns)

In [52]: hier_df
Out[52]:
cty          US                              JP
tenor         1         3         5         1         3
0      0.560145 -1.265934  0.119827 -1.063512  0.332883
1     -2.359419 -0.199543 -1.541996 -0.970736 -1.307030
2      0.286350  0.377984 -0.753887  0.331286  1.349742
3      0.069877  0.246674 -0.011862  1.004812  1.327195

In [53]: hier_df.groupby(level='cty', axis=1).count()
Out[53]:
```

```
cty  JP  US
0     2   3
1     2   3
2     2   3
3     2   3
```

Агрегирование данных

Под агрегированием я обычно понимаю любое преобразование данных, которое порождает скалярные значения из массивов. В примерах выше мы встречали несколько таких преобразований: `mean`, `count`, `min` и `sum`. Вероятно, вам интересно, что происходит при вызове `mean()` для объекта GroupBy. Реализации многих стандартных операций агрегирования, в частности перечисленных в табл. 9.1, оптимизированы для вычисления статистик набора данных *на месте*. Однако необязательно ограничиваться только этими методами. Вы можете придумать собственные способы агрегирования и, кроме того, вызвать любой метод, определенный для сгруппированного объекта. Например, метод `quantile` вычисляет выборочные квантили для объекта Series или столбцов объекта DataFrame[1]:

```
In [54]: df
Out[54]:
      data1     data2 key1 key2
0 -0.204708  1.393406    a  one
1  0.478943  0.092908    a  two
2 -0.519439  0.281746    b  one
3 -0.555730  0.769023    b  two
4  1.965781  1.246435    a  one

In [55]: grouped = df.groupby('key1')

In [56]: grouped['data1'].quantile(0.9)
Out[56]:
key1
a     1.668413
b    -0.523068
```

Хотя в классе GroupBy метод `quantile` не реализован, он есть в классе Series и потому может быть использован. На самом деле, объект GroupBy разбивает Series на части, вызывает `piece.quantile(0.9)` для каждой части, а затем собирает результаты в итоговый объект.

Для использования собственных функций агрегирования передайте функцию, агрегирующую массив, методу `aggregate` или `agg`:

```
In [57]: def peak_to_peak(arr):
   ....:     return arr.max() - arr.min()

In [58]: grouped.agg(peak_to_peak)
Out[58]:
```

[1] Отметим, что метод `quantile` производит линейную интерполяцию, если не существует значения с точно таким процентилем, как указано.

```
         data1     data2
key1
a     2.170488  1.300498
b     0.036292  0.487276
```

Отметим, что некоторые методы, например `describe`, тоже работают, хотя, строго говоря, и не являются операциями агрегирования:

```
In [59]: grouped.describe()
Out[59]:
data1 data2
key1
a    count  3.000000  3.000000
     mean   0.746672  0.910916
     std    1.109736  0.712217
     min   -0.204708  0.092908
     25%    0.137118  0.669671
     50%    0.478943  1.246435
     75%    1.222362  1.319920
     max    1.965781  1.393406
b    count  2.000000  2.000000
     mean  -0.537585  0.525384
     std    0.025662  0.344556
     min   -0.555730  0.281746
     25%   -0.546657  0.403565
     50%   -0.537585  0.525384
     75%   -0.528512  0.647203
     max   -0.519439  0.769023
```

Что здесь произошло, я объясню подробнее в следующем разделе, посвященном групповым операциям и преобразованиям.

Вероятно, вы обратили внимание на то, что пользовательские функции агрегирования работают гораздо медленнее оптимизированных функций из табл. 9.1. Это объясняется большими накладными расходами (на вызовы функций и реорганизацию данных) при построении блоков данных, относящихся к каждой группе.

Таблица 9.1. Оптимизированные функции агрегирования

Имя функции	Описание
`count`	Количество отличных от NA значений в группе
`sum`	Сумма отличных от NA значений
`mean`	Среднее отличных от NA значений
`median`	Медиана отличных от NA значений
`std, var`	Несмещенное (со знаменателем $n - 1$) стандартное отклонение и дисперсия
`min, max`	Минимальное и максимальное отличное от NA значение
`prod`	Произведение отличных от NA значений
`first, last`	Первое и последнее отличное от NA значение

Для иллюстрации более сложных возможностей агрегирования я возьму не столь тривиальный набор данных о ресторанных чаевых. Я получил его с помощью пакета `reshape2` для R; впервые он был приведен в книге Брайана и Смита по экономической статистике 1995 года (и имеется в репозитории этой книге на GitHub). Загрузив его функцией `read_csv`, я добавил столбец с процентом чаевых, `tip_pct`.

```
In [60]: tips = pd.read_csv('ch08/tips.csv')

# Добавить величину чаевых в виде процента от суммы счета
In [61]: tips['tip_pct'] = tips['tip'] / tips['total_bill']

In [62]: tips[:6]
Out[62]:
   total_bill   tip     sex  smoker  day    time   size   tip_pct
0       16.99  1.01  Female      No  Sun   Dinner     2  0.059447
1       10.34  1.66    Male      No  Sun   Dinner     3  0.160542
2       21.01  3.50    Male      No  Sun   Dinner     3  0.166587
3       23.68  3.31    Male      No  Sun   Dinner     2  0.139780
4       24.59  3.61  Female      No  Sun   Dinner     4  0.146808
5       25.29  4.71    Male      No  Sun   Dinner     4  0.186240
```

Применение функций, зависящих от столбца, и нескольких функций

Как мы уже видели, для агрегирования объекта Series или всех столбцов объекта DataFrame достаточно воспользоваться методом `aggregate`, передав ему требуемую функцию, или вызвать метод `mean`, `std` и им подобный. Однако иногда нужно использовать разные функции в зависимости от столбца или сразу несколько функций. К счастью, сделать это совсем нетрудно, что я и продемонстрирую в следующих примерах. Для начала сгруппирую столбец `tips` по значениям `sex` и `smoker`:

```
In [63]: grouped = tips.groupby(['sex', 'smoker'])
```

Отмечу, что в случае описательных статистик типа тех, что приведены в табл. 9.1, можно передать имя функции в виде строки:

```
In [64]: grouped_pct = grouped['tip_pct']

In [65]: grouped_pct.agg('mean')
Out[65]:
sex     smoker
Female  No      0.156921
        Yes     0.182150
Male    No      0.160669
        Yes     0.152771
Name: tip_pct
```

Если вместо этого передать список функций или имен функций, то будет возвращен объект DataFrame, в котором имена столбцов совпадают с именами функций:

```
In [66]: grouped_pct.agg(['mean', 'std', peak_to_peak])
Out[66]:
                   mean       std  peak_to_peak
sex    smoker
Female No     0.156921  0.036421      0.195876
       Yes    0.182150  0.071595      0.360233
Male   No     0.160669  0.041849      0.220186
       Yes    0.152771  0.090588      0.674707
```

Совершенно необязательно соглашаться с именами столбцов, предложенными объектом GroupBy; в частности, все лямбда-функции называются `'<lambda>'`, поэтому различить их затруднительно (можете убедиться сами, распечатав атрибут функции `__name__`). Поэтому если передать список кортежей вида `(name, function)`, то в качестве имени столбца DataFrame будет взят первый элемент кортежа (можно считать, что список 2-кортежей – упорядоченное отображение):

```
In [67]: grouped_pct.agg([('foo', 'mean'), ('bar', np.std)])
Out[67]:
                     foo       bar
sex    smoker
Female No       0.156921  0.036421
       Yes      0.182150  0.071595
Male   No       0.160669  0.041849
       Yes      0.152771  0.090588
```

В случае DataFrame диапазон возможностей шире, поскольку можно задавать список функций, применяемых ко всем столбцам, или разные функции для разных столбцов. Допустим, нам нужно вычислить три одинаковых статистики для столбцов `tip_pct` и `total_bill`:

```
In [68]: functions = ['count', 'mean', 'max']

In [69]: result = grouped['tip_pct', 'total_bill'].agg(functions)

In [70]: result
Out[70]:
               tip_pct                     total_bill
                 count      mean       max       count       mean    max
sex    smoker
Female   No         54  0.156921  0.252672          54  18.105185  35.83
         Yes        33  0.182150  0.416667          33  17.977879  44.30
Male     No         97  0.160669  0.291990          97  19.791237  48.33
         Yes        60  0.152771  0.710345          60  22.284500  50.81
```

Как видите, в результирующем DataFrame имеются иерархические столбцы – точно так же, как было бы, если бы мы агрегировали каждый столбец по отдельности, а потом склеили результаты с помощью метода `concat`, передав ему имена столбцов в качестве аргумента `keys`:

```
In [71]: result['tip_pct']
Out[71]:
               count      mean       max
```

```
sex     smoker
Female  No          54  0.156921  0.252672
        Yes         33  0.182150  0.416667
Male    No          97  0.160669  0.291990
        Yes         60  0.152771  0.710345
```

Как и раньше, можно передавать список кортежей, содержащий желаемые имена:

```
In [72]: ftuples = [('Durchschnitt', 'mean'), ('Abweichung', np.var)]

In [73]: grouped['tip_pct', 'total_bill'].agg(ftuples)
Out[73]:
                    tip_pct                total_bill
                Durchschnitt  Abweichung  Durchschnitt  Abweichung
sex     smoker
Female  No          0.156921    0.001327     18.105185   53.092422
        Yes         0.182150    0.005126     17.977879   84.451517
Male    No          0.160669    0.001751     19.791237   76.152961
        Yes         0.152771    0.008206     22.284500   98.244673
```

Предположим далее, что требуется применить потенциально различные функции к одному или нескольким столбцам. Делается это путем передачи методу `agg` словаря, который содержит отображение имен столбцов на любой из рассмотренных выше объектов, задающих функции:

```
In [74]: grouped.agg({'tip' : np.max, 'size' : 'sum'})
Out[74]:
                size   tip
sex     smoker
Female  No       140   5.2
        Yes       74   6.5
Male    No       263   9.0
        Yes      150  10.0

In [75]: grouped.agg({'tip_pct' : ['min', 'max', 'mean', 'std'],
   ....:              'size' : 'sum'})
Out[75]:
                  tip_pct                                     size
                      min       max      mean       std   sum
sex     smoker
Female  No       0.056797  0.252672  0.156921  0.036421   140
        Yes      0.056433  0.416667  0.182150  0.071595    74
Male    No       0.071804  0.291990  0.160669  0.041849   263
        Yes      0.035638  0.710345  0.152771  0.090588   150
```

Объект DataFrame будет содержать иерархические столбцы, только если хотя бы к одному столбцу было применено несколько функций.

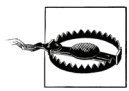
Возврат агрегированных данных в «неиндексированном» виде

Во всех рассмотренных выше примерах агрегированные данные сопровождались индексом, иногда иерархическим, составленным из уникальных встретив-

шихся комбинаций групповых ключей. Такое поведение не всегда желательно, поэтому его можно подавить, передав методу `groupby` параметр `as_index=False`:

```
In [76]: tips.groupby(['sex', 'smoker'], as_index=False).mean()
Out[76]:
      sex  smoker  total_bill       tip      size   tip_pct
0  Female      No   18.105185  2.773519  2.592593  0.156921
1  Female     Yes   17.977879  2.931515  2.242424  0.182150
2    Male      No   19.791237  3.113402  2.711340  0.160669
3    Male     Yes   22.284500  3.051167  2.500000  0.152771
```

Разумеется, для получения данных в таком формате всегда можно вызвать метод `reset_index` результата.



Такое использование `groupby` в общем случае менее гибко; например, результаты с иерархическими столбцами в настоящее время не реализованы, потому что форма результата должна была бы быть почти произвольной.

Групповые операции и преобразования

Агрегирование – лишь одна из групповых операций. Это частный случай более общего класса преобразований, в котором применяемая функция редуцирует одномерный массив в скалярное значение. В этом разделе мы познакомимся с методами `transform` и `apply`, которые позволяют выполнять групповые операции других видов.

Предположим, что требуется добавить столбец в объект DataFrame, содержащий групповые средние для каждого индекса. Для этого можно было сначала выполнить агрегирование, а затем слияние:

```
In [77]: df
Out[77]:
      data1     data2  key1  key2
0 -0.204708  1.393406     a   one
1  0.478943  0.092908     a   two
2 -0.519439  0.281746     b   one
3 -0.555730  0.769023     b   two
4  1.965781  1.246435     a   one

In [78]: k1_means = df.groupby('key1').mean().add_prefix('mean_')

In [79]: k1_means
Out[79]:
      mean_data1  mean_data2
key1
a       0.746672    0.910916
b      -0.537585    0.525384

In [80]: pd.merge(df, k1_means, left_on='key1', right_index=True)
Out[80]:
```

```
      data1     data2  key1  key2  mean_data1  mean_data2
0 -0.204708  1.393406     a   one    0.746672    0.910916
1  0.478943  0.092908     a   two    0.746672    0.910916
4  1.965781  1.246435     a   one    0.746672    0.910916
2 -0.519439  0.281746     b   one   -0.537585    0.525384
3 -0.555730  0.769023     b   two   -0.537585    0.525384
```

Этот способ работает, но ему недостает гибкости. Данную операцию можно рассматривать как преобразование двух столбцов с помощью функции `np.mean`. Рассмотрим еще раз объект DataFrame `people`, встречавшийся выше в этой главе, и воспользуемся методом `transform` объекта GroupBy:

```
In [81]: key = ['one', 'two', 'one', 'two', 'one']

In [82]: people.groupby(key).mean()
Out[82]:
            a         b         c         d         e
one -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
two  0.505275 -0.849512  0.075965  0.834983  0.452620

In [83]: people.groupby(key).transform(np.mean)
Out[83]:
               a         b         c         d         e
Joe    -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
Steve   0.505275 -0.849512  0.075965  0.834983  0.452620
Wes    -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
Jim     0.505275 -0.849512  0.075965  0.834983  0.452620
Travis -0.082032 -1.063687 -1.047620 -0.884358 -0.028309
```

Как легко догадаться, `transform` применяет функцию к каждой группе, а затем помещает результаты в нужные места. Если каждая группа порождает скалярное значение, то оно будет распространено (уложено). Но допустим, что требуется вычесть среднее значение из каждой группы. Для этого напишем функцию `demean` и передадим ее методу `transform`:

```
In [84]: def demean(arr):
   ....:     return arr - arr.mean()

In [85]: demeaned = people.groupby(key).transform(demean)

In [86]: demeaned
Out[86]:
a b c d e
Joe      1.089221 -0.232534  1.322612  1.113271  1.381226
Steve    0.381154 -1.152125 -0.447807  0.834043 -0.891190
Wes     -0.457709       NaN       NaN -0.136869 -0.548778
Jim     -0.381154  1.152125  0.447807 -0.834043  0.891190
Travis  -0.631512  0.232534 -1.322612 -0.976402 -0.832448
```

Легко проверить, что в объекте `demeaned` групповые средние равны нулю:

```
In [87]: demeaned.groupby(key).mean()
Out[87]:
```

```
       a   b  c  d  e
one    0  -0  0  0  0
two   -0   0  0  0  0
```

Как мы увидим в следующем разделе, вычитание среднего для каждой группы можно выполнить также с помощью метода `apply`.

Метод apply: часть общего принципа разделения–применения–объединения

Как и `aggregate`, метод `transform` – более специализированная функция, предъявляющая жесткие требования: переданная ему функция должна возвращать либо скалярное значение, которое можно уложить (как `np.mean`), либо преобразованный массив такого же размера, что исходный. Самым общим из методов класса GroupBy является `apply`, ему мы и посвятим остаток этого раздела. На рис. 9.1 показано, что `apply` разделяет обрабатываемый объект на части, вызывает для каждой части переданную функцию, а затем пытается конкатенировать все части вместе.

Возвращаясь к набору данных о чаевых, предположим, что требуется выбрать первые пять значений `tip_pct` в каждой группе. Прежде всего, нетрудно написать функцию, которая отбирает строки с наибольшими значениями в указанном столбце:

```
In [88]: def top(df, n=5, column='tip_pct'):
   ....:     return df.sort_index(by=column)[-n:]

In [89]: top(tips, n=6)
Out[89]:
     total_bill   tip     sex  smoker  day    time  size   tip_pct
109       14.31  4.00  Female     Yes  Sat  Dinner     2  0.279525
183       23.17  6.50    Male     Yes  Sun  Dinner     4  0.280535
232       11.61  3.39    Male      No  Sat  Dinner     2  0.291990
67         3.07  1.00  Female     Yes  Sat  Dinner     1  0.325733
178        9.60  4.00  Female     Yes  Sun  Dinner     2  0.416667
172        7.25  5.15    Male     Yes  Sun  Dinner     2  0.710345
```

Если теперь сгруппировать, например, по столбцу `smoker`, и вызвать метод `apply`, передав ему эту функцию, то получим следующее:

```
In [90]: tips.groupby('smoker').apply(top)
Out[90]:
            total_bill   tip     sex  smoker   day    time  size   tip_pct
smoker
No      88        24.71  5.85    Male      No  Thur   Lunch     2  0.236746
        185       20.69  5.00    Male      No   Sun  Dinner     5  0.241663
        51        10.29  2.60  Female      No   Sun  Dinner     2  0.252672
        149        7.51  2.00    Male      No  Thur   Lunch     2  0.266312
        232       11.61  3.39    Male      No   Sat  Dinner     2  0.291990
Yes     109       14.31  4.00  Female     Yes   Sat  Dinner     2  0.279525
        183       23.17  6.50    Male     Yes   Sun  Dinner     4  0.280535
```

```
      67        3.07  1.00  Female     Yes   Sat  Dinner     1  0.325733
     178        9.60  4.00  Female     Yes   Sun  Dinner     2  0.416667
     172        7.25  5.15    Male     Yes   Sun  Dinner     2  0.710345
```

Что здесь произошло? Функция `top` вызывается для каждой части DataFrame, после чего результаты склеиваются методом `pandas.concat`, а частям сопоставляются метки, совпадающие с именами групп. Поэтому результат имеет иерархический индекс, внутренний уровень которого содержит индексные значения из исходного объекта DataFrame.

Если передать методу `apply` функцию, которая принимает еще какие-то позиционные или именованные аргументы, то их можно передать вслед за самой функцией:

```
In [91]: tips.groupby(['smoker', 'day']).apply(top, n=1, column='total_bill')
Out[91]:
                total_bill    tip     sex  smoker  day    time size   tip_pct
smoker  day
No      Fri  94       22.75   3.25  Female     No  Fri  Dinner    2  0.142857
        Sat  212      48.33   9.00    Male     No  Sat  Dinner    4  0.186220
        Sun  156      48.17   5.00    Male     No  Sun  Dinner    6  0.103799
       Thur  142      41.19   5.00    Male     No Thur   Lunch    5  0.121389
Yes     Fri  95       40.17   4.73    Male    Yes  Fri  Dinner    4  0.117750
        Sat  170      50.81  10.00    Male    Yes  Sat  Dinner    3  0.196812
        Sun  182      45.35   3.50    Male    Yes  Sun  Dinner    3  0.077178
       Thur  197      43.11   5.00  Female    Yes Thur   Lunch    4  0.115982
```

Это лишь простейшие приемы, а вообще возможности `apply` ограничены только вашей изобретательностью. Что именно делает переданная функция, решать вам, требуется лишь, чтобы она возвращала объект pandas или скалярное значение. Далее в этой главе будут в основном примеры, показывающие, как решать различные задачи с помощью `groupby`.

Вы, наверное, помните, что выше я вызывал метод `describe` от имени объекта GroupBy:

```
In [92]: result = tips.groupby('smoker')['tip_pct'].describe()

In [93]: result
Out[93]:
smoker
No     count   151.000000
       mean      0.159328
       std       0.039910
       min       0.056797
       25%       0.136906
       50%       0.155625
       75%       0.185014
       max       0.291990
Yes    count    93.000000
       mean       0.163196
       std       0.085119
```

```
       min       0.035638
       25%       0.106771
       50%       0.153846
       75%       0.195059
       max       0.710345

In [94]: result.unstack('smoker')
Out[94]:
smoker          No        Yes
count   151.000000  93.000000
mean      0.159328   0.163196
std       0.039910   0.085119
min       0.056797   0.035638
25%       0.136906   0.106771
50%       0.155625   0.153846
75%       0.185014   0.195059
max       0.291990   0.710345
```

Когда от имени GroupBy вызывается метод типа `describe`, на самом деле выполняются такие предложения:

```
f = lambda x: x.describe()
grouped.apply(f)
```

Подавление групповых ключей

В примерах выше мы видели, что у результирующего объекта имеется иерархический индекс, образованный групповыми ключами и индексами каждой части исходного объекта. Создание этого индекса можно подавить, передав методу `groupby` параметр `group_keys=False`:

```
In [95]: tips.groupby('smoker', group_keys=False).apply(top)
Out[95]:
     total_bill   tip     sex  smoker  day    time  size   tip_pct
88         24.71  5.85    Male      No Thur   Lunch     2  0.236746
185        20.69  5.00    Male      No  Sun  Dinner     5  0.241663
51         10.29  2.60  Female      No  Sun  Dinner     2  0.252672
149         7.51  2.00    Male      No Thur   Lunch     2  0.266312
232        11.61  3.39    Male      No  Sat  Dinner     2  0.291990
109        14.31  4.00  Female     Yes  Sat  Dinner     2  0.279525
183        23.17  6.50    Male     Yes  Sun  Dinner     4  0.280535
67          3.07  1.00  Female     Yes  Sat  Dinner     1  0.325733
178         9.60  4.00  Female     Yes  Sun  Dinner     2  0.416667
172         7.25  5.15    Male     Yes  Sun  Dinner     2  0.710345
```

Квантильный и интервальный анализ

Напомним, что в главе 7 шла речь о некоторых средствах библиотеки pandas, в том числе методах `cut` и `qcut`, которые позволяют разложить данные по «ящикам», размер которых задан вами или определяется выборочными квантилями. В сочетании с методом `groupby`, эти методы позволяют очень просто подвергнуть

набор данных интервальному или квантильному анализу. Рассмотрим простой набор случайных данных и раскладывание по интервалам – ящикам – равной длины с помощью `cut`:

```
In [96]: frame = DataFrame({'data1': np.random.randn(1000),
   ....:                    'data2': np.random.randn(1000)})
In [97]: factor = pd.cut(frame.data1, 4)

In [98]: factor[:10]
Out[98]:
Categorical:
array([(-1.23, 0.489], (-2.956, -1.23], (-1.23, 0.489], (0.489, 2.208],
       (-1.23, 0.489], (0.489, 2.208], (-1.23, 0.489], (-1.23, 0.489],
       (0.489, 2.208], (0.489, 2.208]], dtype=object)
Levels (4): Index([(-2.956, -1.23], (-1.23, 0.489], (0.489, 2.208],
                   (2.208, 3.928]], dtype=object)
```

Объект `Factor`, возвращаемый функцией `cut`, можно передать непосредственно `groupby`. Следовательно, набор статистик для столбца `data2` можно вычислить следующим образом:

```
In [99]: def get_stats(group):
   ....:     return {'min': group.min(), 'max': group.max(),
   ....:             'count': group.count(), 'mean': group.mean()}

In [100]: grouped = frame.data2.groupby(factor)

In [101]: grouped.apply(get_stats).unstack()
Out[101]:
                 count      max      mean       min
data1
(-1.23, 0.489]     598 3.260383 -0.002051 -2.989741
(-2.956, -1.23]     95 1.670835 -0.039521 -3.399312
(0.489, 2.208]     297 2.954439  0.081822 -3.745356
(2.208, 3.928]      10 1.765640  0.024750 -1.929776
```

Это были интервалы одинаковой длины, а чтобы вычислить интервалы равного размера на основе выборочных квантилей, нужно использовать функцию `qcut`. Я задам параметр `labels=False`, чтобы получать только номера квантилей:

```
# Вернуть номера квантилей
In [102]: grouping = pd.qcut(frame.data1, 10, labels=False)

In [103]: grouped = frame.data2.groupby(grouping)

In [104]: grouped.apply(get_stats).unstack()
Out[104]:
   count       max       mean        min
0    100  1.670835  -0.049902  -3.399312
1    100  2.628441   0.030989  -1.950098
2    100  2.527939  -0.067179  -2.925113
3    100  3.260383   0.065713  -2.315555
4    100  2.074345  -0.111653  -2.047939
```

```
5    100  2.184810   0.052130  -2.989741
6    100  2.458842  -0.021489  -2.223506
7    100  2.954439  -0.026459  -3.056990
8    100  2.735527   0.103406  -3.745356
9    100  2.377020   0.220122  -2.064111
```

Пример: подстановка зависящих от группы значений вместо отсутствующих

Иногда отсутствующие данные требуется отфильтровать методом `dropna`, а иногда восполнить их, подставив либо фиксированное значение, либо значение, зависящее от данных. Для этой цели предназначен метод `fillna`; вот, например, как можно заменить отсутствующие значения средним:

```
In [105]: s = Series(np.random.randn(6))

In [106]: s[::2] = np.nan

In [107]: s
Out[107]:
0        NaN
1  -0.125921
2        NaN
3  -0.884475
4        NaN
5   0.227290

In [108]: s.fillna(s.mean())
Out[108]:
0  -0.261035
1  -0.125921
2  -0.261035
3  -0.884475
4  -0.261035
5   0.227290
```

А что делать, если подставляемое значение зависит от группы? Как легко догадаться, нужно просто сгруппировать данные и вызвать метод `apply`, передав ему функцию, которая вызывает `fillna` для каждого блока данных. Ниже приведены данные о некоторых штатах США с разделением на восточные и западные:

```
In [109]: states = ['Ohio', 'New York', 'Vermont', 'Florida',
   .....:           'Oregon', 'Nevada', 'California', 'Idaho']

In [110]: group_key = ['East'] * 4 + ['West'] * 4

In [111]: data = Series(np.random.randn(8), index=states)

In [112]: data[['Vermont', 'Nevada', 'Idaho']] = np.nan

In [113]: data
Out[113]:
```

```
Ohio          0.922264
New York     -2.153545
Vermont            NaN
Florida      -0.375842
Oregon        0.329939
Nevada             NaN
California    1.105913
Idaho              NaN

In [114]: data.groupby(group_key).mean()
Out[114]:
East  -0.535707
West   0.717926
```

Чтобы подставить вместо отсутствующих значений групповые средние, нужно поступить так:

```
In [115]: fill_mean = lambda g: g.fillna(g.mean())

In [116]: data.groupby(group_key).apply(fill_mean)
Out[116]:
Ohio         0.922264
New York    -2.153545
Vermont     -0.535707
Florida     -0.375842
Oregon       0.329939
Nevada       0.717926
California   1.105913
Idaho        0.717926
```

Или, возможно, требуется подставлять вместо отсутствующих значений фиксированные, но зависящие от группы:

```
In [117]: fill_values = {'East': 0.5, 'West': -1}

In [118]: fill_func = lambda g: g.fillna(fill_values[g.name])

In [119]: data.groupby(group_key).apply(fill_func)
Out[119]:
Ohio         0.922264
New York    -2.153545
Vermont      0.500000
Florida     -0.375842
Oregon       0.329939
Nevada      -1.000000
California   1.105913
Idaho       -1.000000
```

Пример: случайная выборка и перестановка

Предположим, что требуется произвести случайную выборку (с заменой или без) из большого набора данных для моделирования методом Монте-Карло или какой-то другой задачи. Существуют разные способы выборки, одни более эффек-

тивны, другие – менее. Например, можно взять первые K элементов из множества np.random.permutation(N), где N – размер всего набора данных, а K – размер требуемой выборки. Или вот более занимательный пример – конструирование колоды игральных карт:

```
# Hearts (черви), Spades (пики), Clubs (трефы), Diamonds (бубны)
suits = ['H', 'S', 'C', 'D']
card_val = (range(1, 11) + [10] * 3) * 4
base_names = ['A'] + range(2, 11) + ['J', 'K', 'Q']
cards = []
for suit in ['H', 'S', 'C', 'D']:
   cards.extend(str(num) + suit for num in base_names)

deck = Series(card_val, index=cards)
```

Теперь у нас есть объект Series длины 52, индекс которого содержит названия карт, а значения – ценность карт в блэкджеке и других играх (для простоты я присвоил тузу значение 1).

```
In [121]: deck[:13]
Out[121]:
AH   1
2H   2
3H   3
4H   4
5H   5
6H   6
7H   7
8H   8
9H   9
10H 10
JH  10
KH  10
QH  10
```

На основе сказанного выше сдать пять карт из колоды можно следующим образом:

```
In [122]: def draw(deck, n=5):
   .....:     return deck.take(np.random.permutation(len(deck))[:n])

In [123]: draw(deck)
Out[123]:
AD   1
8C   8
5H   5
KC  10
2C   2
```

Пусть требуется выбрать по две случайных карты из каждой масти. Поскольку масть обозначается последним символом названия карты, то можно произвести по ней группировку и воспользоваться методом apply:

```
In [124]: get_suit = lambda card: card[-1] # last letter is suit

In [125]: deck.groupby(get_suit).apply(draw, n=2)
Out[125]:
C  2C   2
   3C   3
D  KD  10
   8D   8
H  KH  10
   3H   3
S  2S   2
   4S   4

# другой способ
In [126]: deck.groupby(get_suit, group_keys=False).apply(draw, n=2)
Out[126]:
KC  10
JC  10
AD   1
5D   5
5H   5
6H   6
7S   7
KS  10
```

Пример: групповое взвешенное среднее и корреляция

Принцип разделения-применения-объединения, лежащий в основе `groupby`, позволяет легко выразить такие операции между столбцами DataFrame или двумя объектами Series, как вычисление группового взвешенного среднего. В качестве примера возьмем следующий набор данных, содержащий групповые ключи, значения и веса:

```
In [127]: df = DataFrame({'category': ['a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'],
   .....:                 'data': np.random.randn(8),
   .....:                 'weights': np.random.rand(8)})

In [128]: df
Out[128]:
   category       data   weights
0         a   1.561587  0.957515
1         a   1.219984  0.347267
2         a  -0.482239  0.581362
3         a   0.315667  0.217091
4         b  -0.047852  0.894406
5         b  -0.454145  0.918564
6         b  -0.556774  0.277825
7         b   0.253321  0.955905
```

Групповое взвешенное среднее по столбцу `category` равно:

```
In [129]: grouped = df.groupby('category')

In [130]: get_wavg = lambda g: np.average(g['data'], weights=g['weights'])

In [131]: grouped.apply(get_wavg)
Out[131]:
category
a         0.811643
b        -0.122262
```

В качестве не столь тривиального примера рассмотрим набор данных с сайта Yahoo! Finance, содержащий цены дня на некоторые акции и индекс S&P 500 (торговый код `SPX`):

```
In [132]: close_px = pd.read_csv('ch09/stock_px.csv', parse_dates=True, index_col=0)

In [133]: close_px
Out[133]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2214 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00
Data columns:
AAPL    2214  non-null values
MSFT    2214  non-null values
XOM     2214  non-null values
SPX     2214  non-null values
dtypes: float64(4)

In [134]: close_px[-4:]
Out[134]:
              AAPL   MSFT    XOM      SPX
2011-10-11  400.29  27.00  76.27  1195.54
2011-10-12  402.19  26.96  77.16  1207.25
2011-10-13  408.43  27.18  76.37  1203.66
2011-10-14  422.00  27.27  78.11  1224.58
```

Было бы интересно вычислить объект DataFrame, содержащий годовые корреляции между суточной доходностью (вычисленной по процентному изменению) и `SPX`. Ниже показан один из способов решения этой задачи:

```
In [135]: rets = close_px.pct_change().dropna()

In [136]: spx_corr = lambda x: x.corrwith(x['SPX'])

In [137]: by_year = rets.groupby(lambda x: x.year)

In [138]: by_year.apply(spx_corr)
Out[138]:
          AAPL      MSFT       XOM  SPX
2003  0.541124  0.745174  0.661265    1
2004  0.374283  0.588531  0.557742    1
2005  0.467540  0.562374  0.631010    1
2006  0.428267  0.406126  0.518514    1
2007  0.508118  0.658770  0.786264    1
```

```
2008  0.681434  0.804626  0.828303    1
2009  0.707103  0.654902  0.797921    1
2010  0.710105  0.730118  0.839057    1
2011  0.691931  0.800996  0.859975    1
```

Разумеется, ничто не мешает вычислить корреляцию между столбцами:

```
# Годовая корреляция между Apple и Microsoft
In [139]: by_year.apply(lambda g: g['AAPL'].corr(g['MSFT']))
Out[139]:
2003   0.480868
2004   0.259024
2005   0.300093
2006   0.161735
2007   0.417738
2008   0.611901
2009   0.432738
2010   0.571946
2011   0.581987
```

Пример: групповая линейная регрессия

Следуя той же методике, что в предыдущем примере, мы можем применить `groupby` для выполнения более сложного статистического анализа на группах; главное, чтобы функция возвращала объект pandas или скалярное значение. Например, я могу определить функцию `regress` (воспользовавшись эконометрической библиотекой `statsmodels`), которая вычисляет регрессию по обычному методу наименьших квадратов для каждого блока данных:

```
import statsmodels.api as sm
def regress(data, yvar, xvars):
  Y = data[yvar]
  X = data[xvars]
  X['intercept'] = 1.
  result = sm.OLS(Y, X).fit()
  return result.params
```

Теперь для вычисления линейной регрессии `AAPL` от суточного оборота `SPX` нужно написать:

```
In [141]: by_year.apply(regress, 'AAPL', ['SPX'])
Out[141]:
          SPX  intercept
2003  1.195406   0.000710
2004  1.363463   0.004201
2005  1.766415   0.003246
2006  1.645496   0.000080
2007  1.198761   0.003438
2008  0.968016  -0.001110
2009  0.879103   0.002954
2010  1.052608   0.001261
2011  0.806605   0.001514
```

Сводные таблицы и кросс-табуляция

Сводная таблица – это средство обобщения данных, применяемое в электронных таблицах и других аналитических программах. Оно агрегирует таблицу по одному или нескольким ключам и строит другую таблицу, в которой одни групповые ключи расположены в строках, а другие – в столбцах. Библиотека pandas позволяет строить сводные таблицы с помощью описанного выше механизма `groupby` в сочетании с операциями изменения формы с применением иерархического индексирования. В классе DataFrame имеется метод `pivot_table`, а на верхнем уровне – функция `pandas.pivot_table`. Помимо удобного интерфейса к `groupby`, функция `pivot_table` еще умеет добавлять частичные итоги, которые называются *маргиналами*.

Вернемся к набору данных о чаевых и вычислим таблицу групповых средних (тип агрегирования по умолчанию, подразумеваемый `pivot_table`) по столбцам `sex` и `smoker`, расположив их в строках:

```
In [142]: tips.pivot_table(rows=['sex', 'smoker'])
Out[142]:
                size      tip  tip_pct  total_bill
sex    smoker
Female No    2.592593 2.773519 0.156921   18.105185
       Yes   2.242424 2.931515 0.182150   17.977879
Male   No    2.711340 3.113402 0.160669   19.791237
       Yes   2.500000 3.051167 0.152771   22.284500
```

Это можно было бы легко сделать и с помощью `groupby`. Пусть теперь требуется агрегировать только столбцы `tip_pct` и `size`, добавив еще группировку по `day`. Я помещу средние по `smoker` в столбцы таблицы, а по `day` – в строки:

```
In [143]: tips.pivot_table(['tip_pct', 'size'], rows=['sex', 'day'],
   .....:                  cols='smoker')
Out[143]:
             tip_pct               size
smoker            No      Yes       No      Yes
sex    day
Female Fri   0.165296 0.209129 2.500000 2.000000
       Sat   0.147993 0.163817 2.307692 2.200000
       Sun   0.165710 0.237075 3.071429 2.500000
       Thur  0.155971 0.163073 2.480000 2.428571
Male   Fri   0.138005 0.144730 2.000000 2.125000
       Sat   0.162132 0.139067 2.656250 2.629630
       Sun   0.158291 0.173964 2.883721 2.600000
       Thur  0.165706 0.164417 2.500000 2.300000
```

Эту таблицу можно было бы дополнить, включив частичные итоги, для чего следует задать параметр `margins=True`. Тогда будут добавлены строка и столбец с меткой `All`, значениями в которых будут групповые статистики по всем данным на одном уровне. В примере ниже столбцы `All` содержат средние без учета того, является гость курящим или некурящим, а строка `All` – средние по двум уровням группировки:

```
In [144]: tips.pivot_table(['tip_pct', 'size'], rows=['sex', 'day'],
   .....:                  cols='smoker', margins=True)
Out[144]:
                size                     tip_pct
smoker            No      Yes      All       No      Yes      All
sex    day
Female Fri  2.500000 2.000000 2.111111 0.165296 0.209129 0.199388
       Sat  2.307692 2.200000 2.250000 0.147993 0.163817 0.156470
       Sun  3.071429 2.500000 2.944444 0.165710 0.237075 0.181569
       Thur 2.480000 2.428571 2.468750 0.155971 0.163073 0.157525
Male   Fri  2.000000 2.125000 2.100000 0.138005 0.144730 0.143385
       Sat  2.656250 2.629630 2.644068 0.162132 0.139067 0.151577
       Sun  2.883721 2.600000 2.810345 0.158291 0.173964 0.162344
       Thur 2.500000 2.300000 2.433333 0.165706 0.164417 0.165276
All         2.668874 2.408602 2.569672 0.159328 0.163196 0.160803
```

Для применения другой функции агрегирования ее нужно передать в параметре `aggfunc`. Например, передача `'count'` или `len` даст таблицу сопряженности с размерами групп (подсчитывается количество данных в каждой группе):

```
In [145]: tips.pivot_table('tip_pct', rows=['sex', 'smoker'], cols='day',
   .....:                  aggfunc=len, margins=True)
Out[145]:
day         Fri  Sat  Sun  Thur  All
sex    smoker
Female No     2   13   14    25   54
       Yes    7   15    4     7   33
Male   No     2   32   43    20   97
       Yes    8   27   15    10   60
All         19   87   76    62  244
```

Для восполнения отсутствующих комбинаций можно задать параметр `fill_value`:

```
In [146]: tips.pivot_table('size', rows=['time', 'sex', 'smoker'],
   .....:                  cols='day', aggfunc='sum', fill_value=0)
Out[146]:
day                    Fri  Sat  Sun  Thur
time   sex    smoker
Dinner Female No         2   30   43     2
              Yes        8   33   10     0
       Male   No         4   85  124     0
              Yes       12   71   39     0
Lunch  Female No         3    0    0    60
              Yes        6    0    0    17
       Male   No         0    0    0    50
              Yes        5    0    0    23
```

В табл. 9.2 приведена сводка параметров метода `pivot_table`.

Таблица 9.2. Параметры метода `pivot_table`

Параметр	Описание
`values`	Имя (или имена) одного или нескольких столбцов, по которым производится агрегирование. По умолчанию агрегируются все числовые столбцы

Параметр	Описание
rows	Имена столбцов или другие групповые ключи для группировки по строкам результирующей сводной таблицы
cols	Имена столбцов или другие групповые ключи для группировки по столбцам результирующей сводной таблицы
aggfunc	Функция агрегирования или список таких функций; по умолчанию 'mean'. Можно задать произвольную функцию, допустимую в контексте groupby
fill_value	Чем заменять отсутствующие значения в результирующей таблице
margins	Добавлять частичные итоги и общий итог по строкам и столбцам; по умолчанию False

Таблицы сопряженности

Таблица сопряженности – это частный случай сводной таблицы, в которой представлены групповые частоты. Вот канонический пример, взятый со страницы википедии, посвященной таблицам сопряженности:

```
In [150]: data
Out[150]:
   Sample  Gender    Handedness
0       1  Female  Right-handed
1       2  Male     Left-handed
2       3  Female  Right-handed
3       4  Male    Right-handed
4       5  Male     Left-handed
5       6  Male    Right-handed
6       7  Female  Right-handed
7       8  Female   Left-handed
8       9  Male    Right-handed
9      10  Female  Right-handed
```

В ходе анализа-обследования мы могли бы обобщить эти данные по полу и праворукости/леворукости. Для этой цели можно использовать метод pivot_table, но функция pandas.crosstab удобнее:

```
In [151]: pd.crosstab(data.Gender, data.Handedness, margins=True)
Out[151]:
Handedness  Left-handed  Right-handed  All
Gender
Female                1             4    5
Male                  2             3    5
All                   3             7   10
```

Каждый из первых двух аргументов crosstab может быть массивом, объектом Series или списком массивов. Например, в случае данных о чаевых:

```
In [152]: pd.crosstab([tips.time, tips.day], tips.smoker, margins=True)
Out[152]:
```

```
smoker       No Yes  All
time   day
Dinner Fri    3   9   12
       Sat   45  42   87
       Sun   57  19   76
       Thur   1   0    1
Lunch  Fri    1   6    7
       Thur  44  17   61
All         151  93  244
```

Пример: база данных федеральной избирательной комиссии за 2012 год

Федеральная избирательная комиссия США публикует данные о пожертвованиях участникам политических кампаний. Указывается имя жертвователя, род занятий, место работы и сумма пожертвования. Интерес представляет набор данных, относящийся к президентским выборам 2012 года (*http://www.fec.gov/disclosurep/PDownload.do*). На момент написания книги (июль 2012) полный набор данных по всем штатам составляет 150 МБ. Содержащий его CSV-файл `P00000001-ALL.csv` можно скачать с помощью функции `pandas.read_csv`:

```
In [13]: fec = pd.read_csv('ch09/P00000001-ALL.csv')

In [14]: fec
Out[14]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1001731 entries, 0 to 1001730
Data columns:
cmte_id            1001731  non-null values
cand_id            1001731  non-null values
cand_nm            1001731  non-null values
contbr_nm          1001731  non-null values
contbr_city        1001716  non-null values
contbr_st          1001727  non-null values
contbr_zip         1001620  non-null values
contbr_employer    994314   non-null values
contbr_occupation 994433   non-null values
contb_receipt_amt 1001731  non-null values
contb_receipt_dt  1001731  non-null values
receipt_desc       14166    non-null values
memo_cd            92482    non-null values
memo_text          97770    non-null values
form_tp            1001731  non-null values
file_num           1001731  non-null values
dtypes: float64(1), int64(1), object(14)
```

Ниже приведен пример записи в объекте DataFrame:

```
In [15]: fec.ix[123456]
Out[15]:
cmte_id                           C00431445
cand_id                           P80003338
```

```
cand_nm                          Obama, Barack
contbr_nm                          ELLMAN, IRA
contbr_city                              TEMPE
contbr_st                                   AZ
contbr_zip                          852816719
contbr_employer    ARIZONA STATE UNIVERSITY
contbr_occupation                   PROFESSOR
contb_receipt_amt                          50
contb_receipt_dt                    01-DEC-11
receipt_desc                              NaN
memo_cd                                   NaN
memo_text                                 NaN
form_tp                                 SA17A
file_num                               772372
Name:                                  123456
```

Наверное, вы сходу сможете придумать множество способов манипуляции этими данными для извлечения полезной статистики о спонсорах и закономерностях жертвования. Следующие несколько страниц я посвящу различным видам анализа, чтобы проиллюстрировать рассмотренные выше технические приемы.

Как видите, данные не содержат сведений о принадлежности кандидата к политической партии, а эту информацию было бы полезно добавить. Получить список различных кандидатов можно с помощью функции `unique` (отметим, что NumPy подавляет вывод кавычек, в которые заключены строки):

```
In [16]: unique_cands = fec.cand_nm.unique()

In [17]: unique_cands
Out[17]:
array([Bachmann, Michelle, Romney, Mitt, Obama, Barack,
       Roemer, Charles E. 'Buddy' III, Pawlenty, Timothy,
       Johnson, Gary Earl, Paul, Ron, Santorum, Rick, Cain, Herman,
       Gingrich, Newt, McCotter, Thaddeus G, Huntsman, Jon, Perry, Rick],
      dtype=object)

In [18]: unique_cands[2]
Out[18]: 'Obama, Barack'
```

Указать партийную принадлежность проще всего с помощью словаря[2]:

```
parties = {'Bachmann, Michelle': 'Republican',
           'Cain, Herman': 'Republican',
           'Gingrich, Newt': 'Republican',
           'Huntsman, Jon': 'Republican',
           'Johnson, Gary Earl': 'Republican',
           'McCotter, Thaddeus G': 'Republican',
           'Obama, Barack': 'Democrat',
           'Paul, Ron': 'Republican',
           'Pawlenty, Timothy': 'Republican',
```

[2] Здесь сделано упрощающее предположение о том, что Гэри Джонсон – республиканец, хотя впоследствии он стал кандидатом от Либертарианской партии.

```
            'Perry, Rick': 'Republican',
            "Roemer, Charles E. 'Buddy' III": 'Republican',
            'Romney, Mitt': 'Republican',
            'Santorum, Rick': 'Republican'}
```

Далее, применяя этот словарь и метод `map` объектов Series, мы можем построить массив политических партий по именам кандидатов:

```
In [20]: fec.cand_nm[123456:123461]
Out[20]:
123456   Obama, Barack
123457   Obama, Barack
123458   Obama, Barack
123459   Obama, Barack
123460   Obama, Barack
Name: cand_nm

In [21]: fec.cand_nm[123456:123461].map(parties)
Out[21]:
123456   Democrat
123457   Democrat
123458   Democrat
123459   Democrat
123460   Democrat
Name: cand_nm

# Добавить в виде столбца
In [22]: fec['party'] = fec.cand_nm.map(parties)

In [23]: fec['party'].value_counts()
Out[23]:
Democrat     593746
Republican   407985
```

Теперь два замечания касательно подготовки данных. Во-первых, данные включают как пожертвования, так и возвраты (пожертвования со знаком минус):

```
In [24]: (fec.contb_receipt_amt > 0).value_counts()
Out[24]:
True   991475
False   10256
```

Чтобы упростить анализ, я ограничусь только положительными суммами пожертвований:

```
In [25]: fec = fec[fec.contb_receipt_amt > 0]
```

Поскольку главными кандидатам являются Барак Обама и Митт Ромни, я подготовлю также подмножество, содержащее данные о пожертвованиях на их компании:

```
In [26]: fec_mrbo = fec[fec.cand_nm.isin(['Obama, Barack', 'Romney, Mitt'])]
```

Статистика пожертвований по роду занятий и месту работы

Распределение пожертвований по роду занятий – тема, которой посвящено много исследований. Например, юристы (в т. ч. прокуроры) обычно жертвуют в пользу демократов, а руководители предприятий – в пользу республиканцев. Вы вовсе не обязаны верить мне на слово, можете сами проанализировать данные. Для начала решим простую задачу – получим общую статистику пожертвований по роду занятий:

```
In [27]: fec.contbr_occupation.value_counts()[:10]
Out[27]:
RETIRED                                   233990
INFORMATION REQUESTED                      35107
ATTORNEY                                   34286
HOMEMAKER                                  29931
PHYSICIAN                                  23432
INFORMATION REQUESTED PER BEST EFFORTS     21138
ENGINEER                                   14334
TEACHER                                    13990
CONSULTANT                                 13273
PROFESSOR                                  12555
```

Видно, что часто различные занятия на самом деле относятся одной и той же основной профессии, с небольшими вариациями. Ниже показан код, который позволяет произвести очистку, отобразив один род занятий на другой. Обратите внимание на «трюк» с методом `dict.get`, который позволяет «передавать насквозь» занятия, которым ничего не сопоставлено:

```
occ_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'INFORMATION REQUESTED (BEST EFFORTS)' : 'NOT PROVIDED',
   'C.E.O.': 'CEO'
}

# Если ничего не сопоставлено, вернуть x
f = lambda x: occ_mapping.get(x, x)
fec.contbr_occupation = fec.contbr_occupation.map(f)
```

То же самое я проделаю для мест работы:

```
emp_mapping = {
   'INFORMATION REQUESTED PER BEST EFFORTS' : 'NOT PROVIDED',
   'INFORMATION REQUESTED' : 'NOT PROVIDED',
   'SELF' : 'SELF-EMPLOYED',
   'SELF EMPLOYED' : 'SELF-EMPLOYED',
}

# Если ничего не сопоставлено, вернуть x
f = lambda x: emp_mapping.get(x, x)
fec.contbr_employer = fec.contbr_employer.map(f)
```

Теперь можно воспользоваться функцией `pivot_table` для агрегирования данных по партиям и роду занятий, а затем отфильтровать роды занятий, на которые пришлось пожертвований на общую сумму не менее 2 миллионов долларов:

```
In [34]: by_occupation = fec.pivot_table('contb_receipt_amt',
   ....:                                  rows='contbr_occupation',
   ....:                                  cols='party', aggfunc='sum')

In [35]: over_2mm = by_occupation[by_occupation.sum(1) > 2000000]

In [36]: over_2mm
Out[36]:
party                 Democrat         Republican
contbr_occupation
ATTORNEY           11141982.97     7477194.430000
CEO                 2074974.79     4211040.520000
CONSULTANT          2459912.71     2544725.450000
ENGINEER             951525.55     1818373.700000
EXECUTIVE           1355161.05     4138850.090000
HOMEMAKER           4248875.80    13634275.780000
INVESTOR             884133.00     2431768.920000
LAWYER              3160478.87      391224.320000
MANAGER              762883.22     1444532.370000
NOT PROVIDED        4866973.96    20565473.010000
OWNER               1001567.36     2408286.920000
PHYSICIAN           3735124.94     3594320.240000
PRESIDENT           1878509.95     4720923.760000
PROFESSOR           2165071.08      296702.730000
REAL ESTATE          528902.09     1625902.250000
RETIRED            25305116.38    23561244.489999
SELF-EMPLOYED        672393.40     1640252.540000
```

Эти данные проще воспринять в виде графика (параметр `'barh'` означает горизонтальную столбчатую диаграмму, см. рис. 9.2):

```
In [38]: over_2mm.plot(kind='barh')
```

Возможно, вам интересны профессии самых щедрых жертвователей или названия компаний, которые больше всех пожертвовали Обаме или Ромни. Для этого можно сгруппировать данные по имени кандидата, а затем воспользоваться вариантом метода `top`, рассмотренного выше в этой главе:

```
def get_top_amounts(group, key, n=5):
  totals = group.groupby(key)['contb_receipt_amt'].sum()

  # Упорядочить суммы по ключу в порядке убывания
  return totals.order(ascending=False)[-n:]
```

Затем агрегируем по роду занятий и месту работы:

```
In [40]: grouped = fec_mrbo.groupby('cand_nm')

In [41]: grouped.apply(get_top_amounts, 'contbr_occupation', n=7)
```

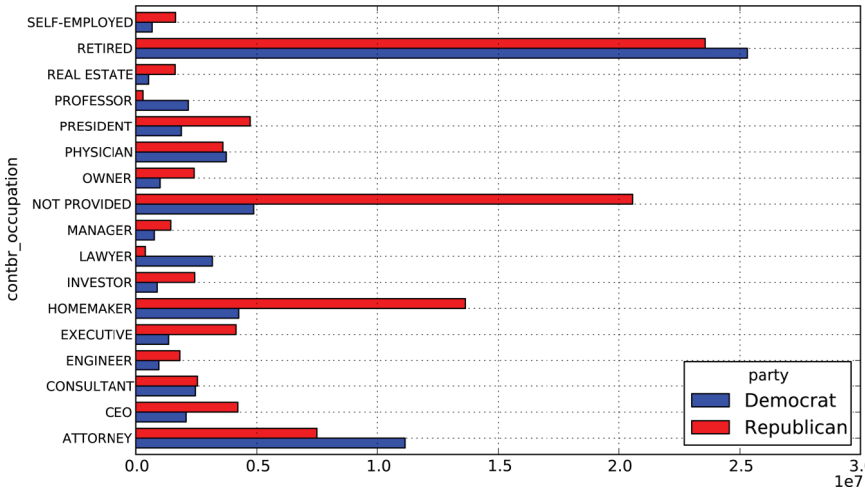


Рис. 9.2. Общая сумма пожертвований по партиям для профессий с максимальной суммой пожертвований

```
Out[41]:
cand_nm contbr_occupation
Obama, Barack  RETIRED              25305116.38
               ATTORNEY             11141982.97
               NOT PROVIDED         4866973.96
               HOMEMAKER            4248875.80
               PHYSICIAN            3735124.94
               LAWYER               3160478.87
               CONSULTANT           2459912.71
Romney, Mitt   RETIRED              11508473.59
               NOT PROVIDED         11396894.84
               HOMEMAKER            8147446.22
               ATTORNEY             5364718.82
               PRESIDENT            2491244.89
               EXECUTIVE            2300947.03
               C.E.O.               1968386.11
Name: contb_receipt_amt

In [42]: grouped.apply(get_top_amounts, 'contbr_employer', n=10)
Out[42]:
cand_nm contbr_employer
Obama, Barack  RETIRED              22694358.85
               SELF-EMPLOYED        18626807.16
               NOT EMPLOYED         8586308.70
               NOT PROVIDED         5053480.37
               HOMEMAKER            2605408.54
               STUDENT              318831.45
               VOLUNTEER            257104.00
               MICROSOFT            215585.36
               SIDLEY AUSTIN LLP    168254.00
               REFUSED              149516.07
Romney, Mitt   NOT PROVIDED         12059527.24
```

```
                RETIRED             11506225.71
                HOMEMAKER           8147196.22
                SELF-EMPLOYED       7414115.22
                STUDENT             496490.94
                CREDIT SUISSE       281150.00
                MORGAN STANLEY      267266.00
                GOLDMAN SACH & CO.  238250.00
                BARCLAYS CAPITAL    162750.00
                H.I.G. CAPITAL      139500.00
Name: contb_receipt_amt
```

Распределение суммы пожертвований по интервалам

Полезный вид анализа данных – дискретизация сумм пожертвований с помощью функции `cut`:

```
In [43]: bins = np.array([0, 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000])

In [44]: labels = pd.cut(fec_mrbo.contb_receipt_amt, bins)

In [45]: labels
Out[45]:
Categorical:contb_receipt_amt
array([(10, 100], (100, 1000], (100, 1000], ..., (1, 10], (10, 100],
       (100, 1000]], dtype=object)
Levels (8): array([(0, 1], (1, 10], (10, 100], (100, 1000], (1000, 10000],
     (10000, 100000], (100000, 1000000], (1000000, 10000000]], dtype=object)
```

Затем можно сгруппировать данные для Обамы и Ромни по имени и метке интервала и построить гистограмму сумм пожертвований:

```
In [46]: grouped = fec_mrbo.groupby(['cand_nm', labels])

In [47]: grouped.size().unstack(0)
Out[47]:
cand_nm                    Obama, Barack         Romney, Mitt
contb_receipt_amt
(0, 1]                               493                   77
(1, 10]                            40070                 3681
(10, 100]                         372280                31853
(100, 1000]                       153991                43357
(1000, 10000]                      22284                26186
(10000, 100000]                        2                    1
(100000, 1000000]                      3                  NaN
(1000000, 10000000]                    4                  NaN
```

Отсюда видно, что Обама получил гораздо больше небольших пожертвований, чем Ромни. Можно также вычислить сумму размеров пожертвований и нормировать распределение по интервалам, чтобы наглядно представить процентную долю пожертвований каждого размера от общего их числа по отдельным кандидатам:

```
In [48]: bucket_sums = grouped.contb_receipt_amt.sum().unstack(0)

In [49]: bucket_sums
Out[49]:
cand_nm                    Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                     318.24         77.00
(1, 10]                    337267.62      29819.66
(10, 100]                  20288981.41    1987783.76
(100, 1000]                54798531.46    22363381.69
(1000, 10000]              51753705.67    63942145.42
(10000, 100000]            59100.00       12700.00
(100000, 1000000]          1490683.08     NaN
(1000000, 10000000]        7148839.76     NaN

In [50]: normed_sums = bucket_sums.div(bucket_sums.sum(axis=1), axis=0)

In [51]: normed_sums
Out[51]:
cand_nm                    Obama, Barack  Romney, Mitt
contb_receipt_amt
(0, 1]                     0.805182       0.194818
(1, 10]                    0.918767       0.081233
(10, 100]                  0.910769       0.089231
(100, 1000]                0.710176       0.289824
(1000, 10000]              0.447326       0.552674
(10000, 100000]            0.823120       0.176880
(100000, 1000000]          1.000000       NaN
(1000000, 10000000]        1.000000       NaN

In [52]: normed_sums[:-2].plot(kind='barh', stacked=True)
```

Я исключил два самых больших интервала, потому что они соответствуют пожертвованиям юридических лиц. Результат показан на рис. 9.3.

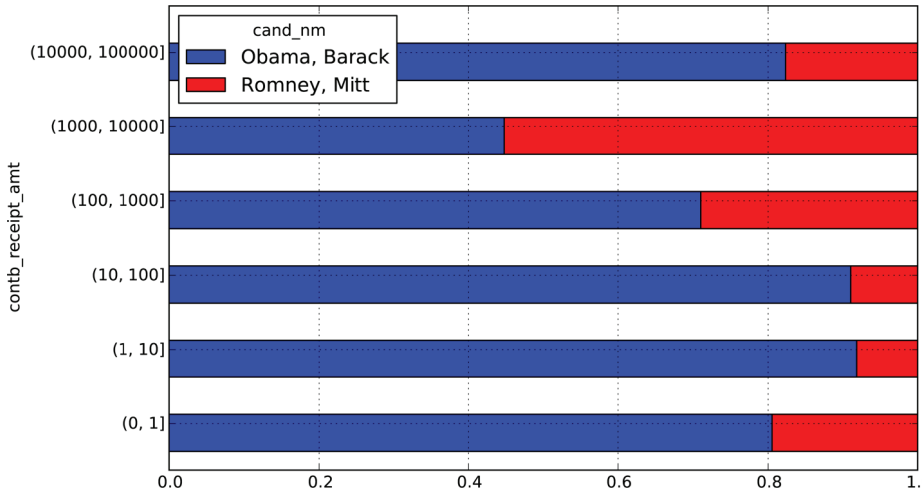


Рис. 9.3. Процентная доля пожертвований каждого размера от общего их числа для обоих кандидатов

Конечно, этот анализ можно уточнить и улучшить во многих направлениях. Например, можно было бы агрегировать пожертвования по имени и почтовому индексу спонсора, чтобы отделить спонсоров, внесших много небольших пожертвований, от тех, кто внес одно или несколько крупных пожертвований. Призываю вас скачать этот набор данных и исследовать его самостоятельно.

Статистика пожертвований по штатам

Агрегирование данных по кандидатам и штатам – вполне рутинная задача:

```
In [53]: grouped = fec_mrbo.groupby(['cand_nm', 'contbr_st'])

In [54]: totals = grouped.contb_receipt_amt.sum().unstack(0).fillna(0)

In [55]: totals = totals[totals.sum(1) > 100000]

In [56]: totals[:10]
Out[56]:
cand_nm     Obama, Barack  Romney, Mitt
contbr_st
AK          281840.15      86204.24
AL          543123.48      527303.51
AR          359247.28      105556.00
AZ          1506476.98     1888436.23
CA          23824984.24    11237636.60
CO          2132429.49     1506714.12
CT          2068291.26     3499475.45
DC          4373538.80     1025137.50
DE          336669.14      82712.00
FL          7318178.58     8338458.81
```

Поделив каждую строку на общую сумму пожертвований, мы получим для каждого кандидата процентную долю от общей суммы, приходящуюся на каждый штат:

```
In [57]: percent = totals.div(totals.sum(1), axis=0)
In [58]: percent[:10]
Out[58]:
cand_nm     Obama, Barack  Romney, Mitt
contbr_st
AK          0.765778       0.234222
AL          0.507390       0.492610
AR          0.772902       0.227098
AZ          0.443745       0.556255
CA          0.679498       0.320502
CO          0.585970       0.414030
CT          0.371476       0.628524
DC          0.810113       0.189887
DE          0.802776       0.197224
FL          0.467417       0.532583
```

Я подумал, что было бы интересно нанести эти данные на карту, следуя идеям из главы 8. Отыскав shape-файл, описывающий границы штатов (*http://*

nationalatlas.gov/atlasftp.html?openChapters=chpbound), и еще немного поизучав пакет matplotlib и дополняющий его пакет `basemap` (в чем мне помогла статья в блоге Томаса Лекока[3], я написал следующую программу нанесения на карту данных об относительном распределении пожертвований:

```
from mpl_toolkits.basemap import Basemap, cm
import numpy as np
from matplotlib import rcParams
from matplotlib.collections import LineCollection
import matplotlib.pyplot as plt

from shapelib import ShapeFile
import dbflib

obama = percent['Obama, Barack']

fig = plt.figure(figsize=(12, 12))
ax = fig.add_axes([0.1,0.1,0.8,0.8])

lllat = 21; urlat = 53; lllon = -118; urlon = -62

m = Basemap(ax=ax, projection='stere',
            lon_0=(urlon + lllon) / 2, lat_0=(urlat + lllat) / 2,
            llcrnrlat=lllat, urcrnrlat=urlat, llcrnrlon=lllon,
            urcrnrlon=urlon, resolution='l')
            m.drawcoastlines()
            m.drawcountries()

shp = ShapeFile('../states/statesp020')
dbf = dbflib.open('../states/statesp020')

for npoly in range(shp.info()[0]):
  # Нарисовать на карте раскрашенные многоугольники
  shpsegs = []
  shp_object = shp.read_object(npoly)
  verts = shp_object.vertices()
  rings = len(verts)
  for ring in range(rings):
     lons, lats = zip(*verts[ring])
     x, y = m(lons, lats)
     shpsegs.append(zip(x,y))
     if ring == 0:
        shapedict = dbf.read_record(npoly)
     name = shapedict['STATE']
  lines = LineCollection(shpsegs,antialiaseds=(1,))

  # словарь state_to_code, сопоставляющий названию штата его код,
  # например'ALASKA' -> 'AK', опущен
  try:
     per = obama[state_to_code[name.upper()]]
  except KeyError:
     continue

  lines.set_facecolors('k')
```

[3] http://www.geophysique.be/2011/01/27/matplotlib-basemap-tutorial-07-shapefiles-unleached/

```
    lines.set_alpha(0.75 * per) # Немного уменьшить процентную долю
    lines.set_edgecolors('k')
    lines.set_linewidth(0.3)
    ax.add_collection(lines)

plt.show()
```

Результат показан на рис. 9.4.

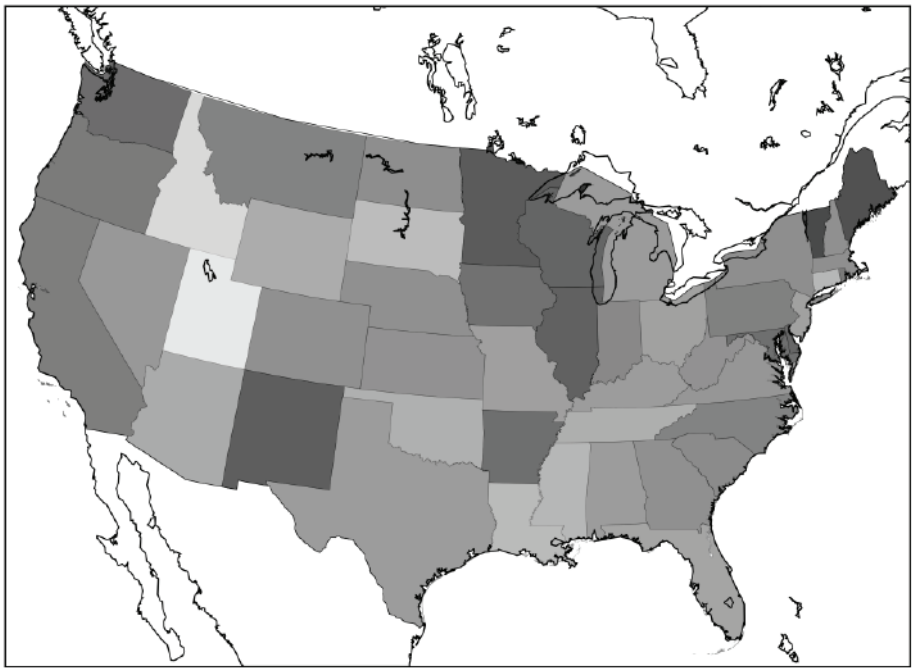


Рис. 9.4. Статистика пожертвований на карте США
(чем область темнее, тем больше она пожертвовала демократической партии)

ГЛАВА 10. Временные ряды

Временные ряды – важная разновидность структурированных данных. Они встречаются во многих областях, в том числе в финансах, экономике, экологии, нейронауках и физике. Любые результаты наблюдений или измерений в разные моменты времени, образуют временной ряд. Для многих временных рядов характерна *фиксированная частота*, т. е. интервалы между соседними точками одинаковы – измерения производятся, например, один раз в 15 секунд, 5 минут или в месяц. Но временные ряды могут быть и *нерегулярными*, когда интервалы времени между соседними точками различаются. Как разметить временной ряд и обращаться к нему, зависит от приложения. Существуют следующие варианты:

- *временные метки*, конкретные моменты времени;
- фиксированные *периоды*, например, январь 2007 или весь 2010 год;
- временные *интервалы*, обозначаемые метками начала и конца; периоды можно считать частными случаями интервалов;
- время эксперимента или истекшее время; каждая временная метка измеряет время, прошедшее с некоторого начального момента. Например, результаты измерения диаметра печенья с момента помещения теста в духовку.

В этой главе меня в основном будут интересовать временные ряды трех первых видов, хотя многие методы применимы и к экспериментальным временным рядам, когда индекс может содержать целые или вещественные значения, обозначающие время, прошедшее с начала эксперимента. Простейший и самый распространенный вид временных рядов – ряды, индексированные временной меткой.

В библиотеке pandas имеется стандартный набор инструментов и алгоритмов для работы с временными рядами. Он позволяет эффективно работать с очень большими рядами, легко строить продольные и поперечные срезы, агрегировать и производить передискретизацию регулярных и нерегулярных временных рядов. Как нетрудно догадаться, многие их этих инструментов особенно полезны в финансовых и эконометрических приложениях, но никто не мешает применять их, например, к анализу журналов сервера.

Некоторые представленные в этой главе средства и код, в особенности относящиеся к периодам, заимствованы из более не поддерживаемой библиотеки `scikits.timeseries`.

Типы данных и инструменты, относящиеся к дате и времени

В стандартной библиотеке Python имеются типы данных для представления даты и времени, а также средства, относящиеся к календарю. Начинать изучение надо с модулей `datetime`, `time` и `calendar`. Особенно широко используется тип `datetime.datetime`, или просто `datetime`:

```
In [317]: from datetime import datetime

In [318]: now = datetime.now()

In [319]: now
Out[319]: datetime.datetime(2012, 8, 4, 17, 9, 21, 832092)

In [320]: now.year, now.month, now.day
Out[320]: (2012, 8, 4)
```

В объекте типа `datetime` хранятся дата и время с точностью до микросекунды. Класс `datetime.timedelta` представляет интервал времени между двумя объектами `datetime`:

```
In [321]: delta = datetime(2011, 1, 7) - datetime(2008, 6, 24, 8, 15)

In [322]: delta
Out[322]: datetime.timedelta(926, 56700)

In [323]: delta.days              In [324]: delta.seconds
Out[323]: 926                     Out[324]: 56700
```

Можно прибавить (или вычесть) объект `timedelta` или его произведение на целое число к объекту `datetime` и получить в результате новый объект того же типа, представляющий соответственно сдвинутый момент времени:

```
In [325]: from datetime import timedelta

In [326]: start = datetime(2011, 1, 7)

In [327]: start + timedelta(12)
Out[327]: datetime.datetime(2011, 1, 19, 0, 0)

In [328]: start - 2 * timedelta(12)
Out[328]: datetime.datetime(2010, 12, 14, 0, 0)
```

Сводка типов данных в модуле `datetime` приведена в табл. 10.1. Хотя в этой главе речь пойдет преимущественно о типах данных в pandas и высокоуровневых операциях с временными рядами, вы без сомнения встретите основанные на `datetime` типы и во многих других приложениях, написанных на Python.

Таблица 10.1. Типы в модуле datetime

Тип	Описание
`date`	Хранит дату (год, месяц, день) по григорианскому календарю
`time`	Хранит время суток (часы, минуты, секунды и микросекунды)
`datetime`	Хранит дату и время
`timedelta`	Представляет разность между двумя значениями типа `datetime` (дни, секунды и микросекунды)

Преобразование между строкой и datetime

Объекты типа `datetime` и входящего в pandas типа `Timestamp`, с которым мы вскоре познакомимся, можно представить в виде отформатированной строки с помощью метода `str` или `strftime`, которому передается спецификация формата:

```
In [329]: stamp = datetime(2011, 1, 3)

In [330]: str(stamp) In [331]: stamp.strftime('%Y-%m-%d')
Out[330]: '2011-01-03 00:00:00' Out[331]: '2011-01-03'
```

Полный перечень форматных кодов приведен в табл. 10.2. Те же самые коды используются для преобразования строк в даты методом `datetime.strptime`:

```
In [332]: value = '2011-01-03'

In [333]: datetime.strptime(value, '%Y-%m-%d')
Out[333]: datetime.datetime(2011, 1, 3, 0, 0)

In [334]: datestrs = ['7/6/2011', '8/6/2011']

In [335]: [datetime.strptime(x, '%m/%d/%Y') for x in datestrs]
Out[335]: [datetime.datetime(2011, 7, 6, 0, 0), datetime.datetime(2011, 8, 6, 0, 0)]
```

Метод `datetime.strptime` прекрасно работает, когда формат даты известен. Однако каждый раз задавать формат даты, особенно общеупотребительный, надоедает. В таком случае можно воспользоваться методом `parser.parse` из стороннего пакета `dateutil`:

```
In [336]: from dateutil.parser import parse

In [337]: parse('2011-01-03')
Out[337]: datetime.datetime(2011, 1, 3, 0, 0)
```

`dateutil` умеет разбирать практически любое представление даты, понятное человеку:

```
In [338]: parse('Jan 31, 1997 10:45 PM')
Out[338]: datetime.datetime(1997, 1, 31, 22, 45)
```

Если, как бывает в других локалях, день предшествует месяцу, то следует задать параметр `dayfirst=True`:

```
In [339]: parse('6/12/2011', dayfirst=True)
Out[339]: datetime.datetime(2011, 12, 6, 0, 0)
```

Библиотека pandas, вообще говоря, ориентирована на работу с массивами дат, используемых как в качестве осевого индекса, так и столбца в DataFrame. Метод `to_datetime` разбирает различные представления даты. Стандартные форматы, например ISO8601, разбираются очень быстро.

```
In [340]: datestrs
Out[340]: ['7/6/2011', '8/6/2011']

In [341]: pd.to_datetime(datestrs)
Out[341]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-07-06 00:00:00, 2011-08-06 00:00:00]
Length: 2, Freq: None, Timezone: None
```

Кроме того, этот метод умеет обрабатывать значения, которые следует считать отсутствующими (`None`, пустая строка и т. д.):

```
In [342]: idx = pd.to_datetime(datestrs + [None])

In [343]: idx
Out[343]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-07-06 00:00:00, ..., NaT]
Length: 3, Freq: None, Timezone: None

In [344]: idx[2]
Out[344]: NaT

In [345]: pd.isnull(idx)
Out[345]: array([False, False, True], dtype=bool)
```

`NaT` (Not a Time – не время) – применяемое в pandas значение для индикации отсутствующей временной метки.

Таблица 10.2. Спецификации формата даты в классе datetime (совместима со стандартом ISO C89)

Спецификатор	Описание
`%Y`	Год с четырьмя цифрами
`%y`	Год с двумя цифрами
`%m`	Номер месяца с двумя цифрами [01, 12]
`%d`	Номер дня с двумя цифрами [01, 31]
`%H`	Час (в 24-часовом формате) [00, 23]

Спецификатор	Описание
%h	Час (в 12-часовом формате) [01, 12]
%M	Минута с двумя цифрами [01, 59]
%S	Секунда [00, 61] (секунды 60 и 61 високосные)
%w	День недели в виде целого числа [0 (воскресенье), 6]
%U	Номер недели в году [00, 53]. Первым днем недели считается воскресенье, а дни, предшествующие первому воскресенью, относятся к «неделе 0»
%W	Номер недели в году [00, 53]. Первым днем недели считается понедельник, а дни, предшествующие первому понедельнику, относятся к «неделе 0»
%z	Часовой пояс UTC в виде +HHMM или -HHMM; пустая строка, если часовой пояс не учитывается
%F	Сокращение для %Y-%m-%d, например 2012-4-18
%D	Сокращение для %m/%d/%y, например 04/18/2012

Класс `dateutil.parser` – полезный, но не идеальный инструмент. В частности, он распознает строки, которые не на всякий взгляд являются датами. Например, строка '42' будет разобрана как текущая календарная дата в 2042 году.

У объектов `datetime` имеется также ряд зависимых от локали параметров форматирования для других стран и языков. Например, сокращенные названия месяцев в системе с немецкой или французской локалью будут не такие, как в системе с английской локалью.

Таблица 10.3. Спецификации формата даты, зависящие от локали

Спецификатор	Описание
%a	Сокращенное название дня недели
%A	Полное название дня недели
%b	Сокращенное название месяца
%B	Полное название месяца
%c	Полная дата и время, например «Tue 01 May 2012 04:20:57 PM»
%p	Локализованный эквивалент AM или PM
%x	Дата в формате, соответствующем локали. Например, в США 1 мая 2012 будет представлена в виде «05/01/2012»
%X	Время в формате, соответствующем локали, например «04:24:12 PM»

Основы работы с временными рядами

Самый простой вид временного ряда в pandas – объект Series, индексированный временными метками, которые часто представляются внешними по отношению к pandas Python-строками или объектами `datetime`:

```
In [346]: from datetime import datetime

In [347]: dates = [datetime(2011, 1, 2), datetime(2011, 1, 5), datetime(2011, 1, 7),
   .....:          datetime(2011, 1, 8), datetime(2011, 1, 10), datetime(2011, 1, 12)]

In [348]: ts = Series(np.random.randn(6), index=dates)

In [349]: ts
Out[349]:
2011-01-02   0.690002
2011-01-05   1.001543
2011-01-07  -0.503087
2011-01-08  -0.622274
2011-01-10  -0.921169
2011-01-12  -0.726213
```

Под капотом объекты `datetime` помещаются в объект типа `DatetimeIndex`, а переменная `ts` получает тип `TimeSeries`:

```
In [350]: type(ts)
Out[350]: pandas.core.series.TimeSeries

In [351]: ts.index
Out[351]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-02 00:00:00, ..., 2011-01-12 00:00:00]
Length: 6, Freq: None, Timezone: None
```

Необязательно использовать конструктор `TimeSeries` явно; если объект Series создается с индексом типа `DatetimeIndex`, то pandas знает, что это временной ряд.

Как и для других объектов Series, арифметические операции над временными рядами с различными индексами автоматически приводят к выравниванию дат:

```
In [352]: ts + ts[::2]
Out[352]:
2011-01-02   1.380004
2011-01-05        NaN
2011-01-07  -1.006175
2011-01-08        NaN
2011-01-10  -1.842337
2011-01-12        NaN
```

В pandas временные метки хранятся в типе данных NumPy `datetime64` с наносекундным разрешением:

```
In [353]: ts.index.dtype
Out[353]: dtype('datetime64[ns]')
```

Скалярные значения в индексе `DatetimeIndex` – это объекты pandas типа `Timestamp`:

```
In [354]: stamp = ts.index[0]
In [355]: stamp
Out[355]: <Timestamp: 2011-01-02 00:00:00>
```

Объект `Timestamp` можно использовать всюду, где допустим объект `datetime`. Кроме того, в нем можно хранить информацию о частоте (если имеется) и он умеет преобразовывать часовые пояса и производить другие манипуляции. Подробнее об этом будет рассказано ниже.

Индексирование, выборка, подмножества

TimeSeries – подкласс Series и потому ведет себя точно так же по отношению к индексированию и выборке данных по метке:

В качестве дополнительного удобства можно передать строку, допускающую интерпретацию в виде даты:

```
In [358]: ts['1/10/2011'] In [359]: ts['20110110']
Out[358]: -0.92116860801301081 Out[359]: -0.92116860801301081
```

Для выборки срезов из длинных временных рядов можно передать только год или год и месяц:

```
In [360]: longer_ts = Series(np.random.randn(1000),
   .....:                    index=pd.date_range('1/1/2000', periods=1000))

In [361]: longer_ts
Out[361]:
2000-01-01   0.222896
2000-01-02   0.051316
2000-01-03  -1.157719
2000-01-04   0.816707
...
2002-09-23  -0.395813
2002-09-24  -0.180737
2002-09-25   1.337508
2002-09-26  -0.416584
Freq: D, Length: 1000

In [362]: longer_ts['2001']
Out[362]:
2001-01-01  -1.499503
2001-01-02   0.545154
2001-01-03   0.400823
2001-01-04  -1.946230
...
2001-12-28  -1.568139

In [363]: longer_ts['2001-05']
2001-05-01   1.662014
2001-05-02  -1.189203
2001-05-03   0.093597
2001-05-04  -0.539164
...
2001-05-28  -0.683066
```

```
2001-12-29  -0.900887              2001-05-29  -0.950313
2001-12-30   0.652346              2001-05-30   0.400710
2001-12-31   0.871600              2001-05-31  -0.126072
Freq: D, Length: 365 Freq: D, Length: 31
```

Выборка срезов с помощью дат работает, как и в обычных объектах Series:

```
In [364]: ts[datetime(2011, 1, 7):]
Out[364]:
2011-01-07  -0.503087
2011-01-08  -0.622274
2011-01-10  -0.921169
2011-01-12  -0.726213
```

Поскольку временные ряды обычно упорядочены хронологически, при формировании срезов можно указывать временные метки, отсутствующие в самом ряду, т. е. выполнять запрос по диапазону:

```
In [365]: ts                 In [366]: ts['1/6/2011':'1/11/2011']
Out[365]:                    Out[366]:
2011-01-02   0.690002        2011-01-07  -0.503087
2011-01-05   1.001543        2011-01-08  -0.622274
2011-01-07  -0.503087        2011-01-10  -0.921169
2011-01-08  -0.622274
2011-01-10  -0.921169
2011-01-12  -0.726213
```

Как и раньше, можно задать дату в виде строки, объекта `datetime` или `Timestamp`. Напомню, что такое формирование среза порождает представление исходного временного ряда, как и для массивов NumPy. Существует эквивалентный метод экземпляра `truncate`, который возвращает срез TimeSeries между двумя датами:

```
In [367]: ts.truncate(after='1/9/2011')
Out[367]:
2011-01-02   0.690002
2011-01-05   1.001543
2011-01-07  -0.503087
2011-01-08  -0.622274
```

Все вышеперечисленное справедливо и для индексирования объекта DataFrame по строкам:

```
In [368]: dates = pd.date_range('1/1/2000', periods=100, freq='W-WED')
In [369]: long_df = DataFrame(np.random.randn(100, 4),
   .....:                     index=dates,
   .....:                  columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [370]: long_df.ix['5-2001']
Out[370]:
             Colorado      Texas   New York       Ohio
2001-05-02   0.943479  -0.349366   0.530412  -0.508724
2001-05-09   0.230643  -0.065569  -0.248717  -0.587136
```

```
2001-05-16  -1.022324   1.060661   0.954768  -0.511824
2001-05-23  -1.387680   0.767902  -1.164490   1.527070
2001-05-30   0.287542   0.715359  -0.345805   0.470886
```

Временные ряды с неуникальными индексами

В некоторых приложениях бывает, что несколько результатов измерений имеют одну и ту же временную метку, например:

```
In [371]: dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000',
   .....:                           '1/3/2000'])

In [372]: dup_ts = Series(np.arange(5), index=dates)

In [373]: dup_ts
Out[373]:
2000-01-01   0
2000-01-02   1
2000-01-02   2
2000-01-02   3
2000-01-03   4
```

Узнать о том, что индекс не уникален, можно, опросив его свойство `is_unique`:

```
In [374]: dup_ts.index.is_unique
Out[374]: False
```

При доступе к такому временному ряду по индексу будет возвращено либо скалярное значение, либо срез – в зависимости от того, является временная метка уникальной или нет:

```
In [375]: dup_ts['1/3/2000'] # метка уникальна
Out[375]: 4

In [376]: dup_ts['1/2/2000'] # метка повторяется
Out[376]:
2000-01-02   1
2000-01-02   2
2000-01-02   3
```

Пусть требуется агрегировать данные с неуникальными временными метками. Одно из возможных решений – воспользоваться методом `groupby` с параметром `level=0` (единственный уровень индексирования!):

```
In [377]: grouped = dup_ts.groupby(level=0)

In [378]: grouped.mean()          In [379]: grouped.count()
Out[378]:                         Out[379]:
2000-01-01   0                    2000-01-01   1
2000-01-02   2                    2000-01-02   3
2000-01-03   4                    2000-01-03   1
```

Диапазоны дат, частоты и сдвиг

Вообще говоря, временные ряды pandas не предполагаются регулярными, т. е. частота в них не фиксирована. Для многих приложений это вполне приемлемо. Но иногда желательно работать с постоянной частотой, например, день, месяц, 15 минут, даже если для этого приходится вставлять в ряд отсутствующие значения. По счастью, pandas поддерживает полный набор частот и средства для передискретизации, выведения частот и генерации диапазонов дат с фиксированной частотой. Например, временной ряд из нашего примера можно преобразовать в ряд с частотой один день с помощью метода `resample`:

```
In [380]: ts
Out[380]:
2011-01-02   0.690002
2011-01-05   1.001543
2011-01-07  -0.503087
2011-01-08  -0.622274
2011-01-10  -0.921169
2011-01-12  -0.726213

Freq: D
```

```
In [381]: ts.resample('D')
Out[381]:
2011-01-02   0.690002
2011-01-03        NaN
2011-01-04        NaN
2011-01-05   1.001543
2011-01-06        NaN
2011-01-07  -0.503087
2011-01-08  -0.622274
2011-01-09        NaN
2011-01-10  -0.921169
2011-01-11        NaN
2011-01-12  -0.726213
```

Преобразование частоты, или *передискретизация* – настолько обширная тема, что мы посвятим ей отдельный раздел ниже. А сейчас я покажу, как работать с базовой частотой и кратными ей.

Генерация диапазонов дат

Раньше я уже пользовался методом `pandas.date_range` без объяснений, и вы, наверное, догадались, что он порождает объект `DatetimeIndex` указанной длины с определенной частотой:

```
In [382]: index = pd.date_range('4/1/2012', '6/1/2012')

In [383]: index
Out[383]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-04-01 00:00:00, ..., 2012-06-01 00:00:00]
Length: 62, Freq: D, Timezone: None
```

По умолчанию метод `date_range` генерирует временные метки с частотой один день. Если вы передаете ему только начальную или конечную дату, то должны задать также количество генерируемых периодов:

```
In [384]: pd.date_range(start='4/1/2012', periods=20)
Out[384]:
<class 'pandas.tseries.index.DatetimeIndex'>
```

```
[2012-04-01 00:00:00, ..., 2012-04-20 00:00:00]
Length: 20, Freq: D, Timezone: None

In [385]: pd.date_range(end='6/1/2012', periods=20)
Out[385]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-13 00:00:00, ..., 2012-06-01 00:00:00]
Length: 20, Freq: D, Timezone: None
```

Начальная и конечная дата определяют строгие границы для сгенерированного индекса по датам. Например, если требуется индекс по датам, содержащий последний рабочий день каждого месяца, то следует передать в качестве частоты значение `'BM'`, и тогда будут включены только даты, попадающие внутрь или на границу интервала:

```
In [386]: pd.date_range('1/1/2000', '12/1/2000', freq='BM')
Out[386]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-31 00:00:00, ..., 2000-11-30 00:00:00]
Length: 11, Freq: BM, Timezone: None
```

По умолчанию метод `date_range` сохраняет время (если оно было задано) начальной и конечной временной метки:

```
In [387]: pd.date_range('5/2/2012 12:56:31', periods=5)
Out[387]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-02 12:56:31, ..., 2012-05-06 12:56:31]
Length: 5, Freq: D, Timezone: None
```

Иногда начальная или конечная дата содержит время, но требуется сгенерировать *нормализованный* набор временных меток, в которых время совпадает с полуночью. Для этого задайте параметр `normalize`:

```
In [388]: pd.date_range('5/2/2012 12:56:31', periods=5, normalize=True)
Out[388]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-02 00:00:00, ..., 2012-05-06 00:00:00]
Length: 5, Freq: D, Timezone: None
```

Частоты и смещения дат

Частота в pandas состоит из *базовой частоты* и кратности. Базовая частота обычно обозначается строкой, например, `'M'` означает раз в месяц, а `'H'` – раз в час. Для каждой базовой частоты определен объект, называемый *смещением даты* (date offset). Так, частоту «раз в час» можно представить классом `Hour`:

```
In [389]: from pandas.tseries.offsets import Hour, Minute

In [390]: hour = Hour()

In [391]: hour
Out[391]: <1 Hour>
```

Для определения кратности смещения нужно задать целое число:

```
In [392]: four_hours = Hour(4)

In [393]: four_hours
Out[393]: <4 Hours>
```

В большинстве приложений не приходится создавать такие объекты явно, достаточно использовать их строковые обозначения вида `'H'` или `'4H'`. Наличие целого числа перед базовой частотой создает кратную частоту:

```
In [394]: pd.date_range('1/1/2000', '1/3/2000 23:59', freq='4h')
Out[394]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-03 20:00:00]
Length: 18, Freq: 4H, Timezone: None
```

Операция сложения позволяет объединить несколько смещений:

```
In [395]: Hour(2) + Minute(30)
Out[395]: <150 Minutes>
```

Можно также задать частоту в виде строки `'2h30min'`, что приводит к тому же результату, что и выше:

```
In [396]: pd.date_range('1/1/2000', periods=10, freq='1h30min')
Out[396]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-01 13:30:00]
Length: 10, Freq: 90T, Timezone: None
```

Некоторые частоты описывает неравноотстоящие моменты времени. Например, значение частот `'M'` (конец календарного месяца) и `'BM'` (последний рабочий день месяца) зависят от числа дней в месяце, а в последнем случае также от того, заканчивается месяц рабочим или выходным днем. За неимением лучшего термина я называю такие смещения *привязанными*.

В табл. 10.4 перечислены имеющиеся в pandas коды частот и классы смещений дат.

Пользователь может определить собственный класс частоты для реализации логики работы с датами, отсутствующей в pandas, однако подробности этого процесса выходят за рамки книги.

Таблица 10.4. Базовые частоты временных рядов

Обозначение	Тип смещения	Описание
`D`	`Day`	Ежедневно
`B`	`BusinessDay`	Каждый рабочий день
`H`	`Hour`	Ежечасно

Обозначение	Тип смещения	Описание
T или min	Minute	Ежеминутно
S	Second	Ежесекундно
L или ms	Milli	Каждую миллисекунду
U	Micro	Каждую микросекунду
M	MonthEnd	Последний календарный день месяца
BM	BusinessMonthEnd	Последний рабочий день месяца
MS	MonthBegin	Первый календарный день месяца
BMS	BusinessMonthBegin	Первый рабочий день месяца
W-MON, W-TUE, ...	Week	Еженедельно в указанный день: MON, TUE, WED, THU, FRI, SAT, SUN
WOM-1MON, WOM-2MON, ...	WeekOfMonth	Указанный день первой, второй, третьей или четвертой недели месяца. Например, WOM-3FRI означает третью пятницу каждого месяца
Q-JAN, Q-FEB, ...	QuarterEnd	Ежеквартально с привязкой к последнему календарному дню каждого месяца, считая, что год заканчивается в указанном месяце: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
BQ-JAN, BQ-FEB, ...	BusinessQuarterEnd	Ежеквартально с привязкой к последнему рабочему дню каждого месяца, считая, что год заканчивается в указанном месяце
QS-JAN, QS-FEB, ...	QuarterBegin	Ежеквартально с привязкой к первому календарному дню каждого месяца, считая, что год заканчивается в указанном месяце
BQS-JAN, BQS-FEB, ...	BusinessQuarterBegin	Ежеквартально с привязкой к первому рабочему дню каждого месяца, считая, что год заканчивается в указанном месяце
A-JAN, A-FEB, ...	YearEnd	Ежегодно с привязкой к последнему календарному дню указанного месяца: JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
BA-JAN, BA-FEB, ...	BusinessYearEnd	Ежегодно с привязкой к последнему рабочему дню указанного месяца
AS-JAN, AS-FEB, ...	YearBegin	Ежегодно с привязкой к первому календарному дню указанного месяца
BAS-JAN, BAS-FEB, ...	BusinessYearBegin	Ежегодно с привязкой к первому рабочему дню указанного месяца

Даты, связанные с неделей месяца

Полезный класс частот – «неделя месяца», обозначается строкой, начинающейся с WOM. Он позволяет получить, например, третью пятницу каждого месяца:

```
In [397]: rng = pd.date_range('1/1/2012', '9/1/2012', freq='WOM-3FRI')
In [398]: list(rng)
Out[398]:
[<Timestamp: 2012-01-20 00:00:00>,
 <Timestamp: 2012-02-17 00:00:00>,
 <Timestamp: 2012-03-16 00:00:00>,
 <Timestamp: 2012-04-20 00:00:00>,
 <Timestamp: 2012-05-18 00:00:00>,
 <Timestamp: 2012-06-15 00:00:00>,
 <Timestamp: 2012-07-20 00:00:00>,
 <Timestamp: 2012-08-17 00:00:00>]
```

Торговцы опционами на акции компаний США узнают здесь стандартные даты окончания месяца.

Сдвиг данных (с опережением и с запаздыванием)

Под «сдвигом» понимается перемещение данных назад и вперед по временной оси. У объектов Series и DataFrame имеется метод shift для «наивного» сдвига в обе стороны без модификации индекса:

```
In [399]: ts = Series(np.random.randn(4),
   .....: index=pd.date_range('1/1/2000', periods=4, freq='M'))

In [400]: ts            In [401]: ts.shift(2)       In [402]: ts.shift(-2)
Out[400]:                Out[401]:                   Out[402]:
2000-01-31  0.575283     2000-01-31       NaN        2000-01-31  1.814582
2000-02-29  0.304205     2000-02-29       NaN        2000-02-29  1.634858
2000-03-31  1.814582     2000-03-31  0.575283        2000-03-31       NaN
2000-04-30  1.634858     2000-04-30  0.304205        2000-04-30       NaN
Freq: M Freq: M Freq: M
```

Типичное применение shift – вычисление относительных изменений временного ряда или нескольких временных рядов и представление их в виде столбцов объекта DataFrame. Это выражается следующим образом:

```
ts / ts.shift(1) - 1
```

Поскольку наивный сдвиг не изменяет индекс, некоторые данные отбрасываются. Но если известна частота, то ее можно передать методу shift, чтобы сдвинуть вперед временные метки, а не сами данные:

```
In [403]: ts.shift(2, freq='M')
Out[403]:
2000-03-31   0.575283
2000-04-30   0.304205
2000-05-31   1.814582
2000-06-30   1.634858
Freq: M
```

Можно указывать и другие частоты, что позволяет очень гибко смещать данные в прошлое и в будущее:

```
In [404]: ts.shift(3, freq='D')
Out[404]:
2000-02-03   0.575283
2000-03-03   0.304205
2000-04-03   1.814582
2000-05-03   1.634858

In [405]: ts.shift(1, freq='3D')
Out[405]:
2000-02-03   0.575283
2000-03-03   0.304205
2000-04-03   1.814582
2000-05-03   1.634858

In [406]: ts.shift(1, freq='90T')
Out[406]:
2000-01-31 01:30:00   0.575283
2000-02-29 01:30:00   0.304205
2000-03-31 01:30:00   1.814582
2000-04-30 01:30:00   1.634858
```

Сдвиг дат с помощью смещений

Совместно с объектами `datetime` и `Timestamp` можно использовать также смещения дат pandas:

```
In [407]: from pandas.tseries.offsets import Day, MonthEnd

In [408]: now = datetime(2011, 11, 17)

In [409]: now + 3 * Day()
Out[409]: datetime.datetime(2011, 11, 20, 0, 0)
```

В случае привязанного смещения, например `MonthEnd`, первое сложение с ним *продвинет* дату до следующей даты с соответствующей привязкой:

```
In [410]: now + MonthEnd()
Out[410]: datetime.datetime(2011, 11, 30, 0, 0)

In [411]: now + MonthEnd(2)
Out[411]: datetime.datetime(2011, 12, 31, 0, 0)
```

Привязанные смещения можно использовать и для явного сдвига даты вперед и назад с помощью методов `rollforward` и `rollback` соответственно:

```
In [412]: offset = MonthEnd()

In [413]: offset.rollforward(now)
Out[413]: datetime.datetime(2011, 11, 30, 0, 0)

In [414]: offset.rollback(now)
Out[414]: datetime.datetime(2011, 10, 31, 0, 0)
```

У смещений дат есть интересное применение совместно с функцией `groupby`:

```
In [415]: ts = Series(np.random.randn(20),
   .....:             index=pd.date_range('1/15/2000', periods=20, freq='4d'))

In [416]: ts.groupby(offset.rollforward).mean()
```

```
Out[416]:
2000-01-31   -0.448874
2000-02-29   -0.683663
2000-03-31    0.251920
```

Разумеется, проще и быстрее добиться того же результата с помощью метода `resample` (подробнее о нем будет сказано ниже):

```
In [417]: ts.resample('M', how='mean')
Out[417]:
2000-01-31   -0.448874
2000-02-29   -0.683663
2000-03-31    0.251920
Freq: M
```

Часовые пояса

Работа с часовым поясами традиционно считается одной из самых неприятных сторон манипулирования временными рядами. В частности, распространенным источником затруднений является переход на летнее время. Поэтому многие пользователи предпочитают иметь дело с временными рядами в *координированном универсальном времени* (*UTC*), которое пришло на смену Гринвичскому времени и теперь является международным стандартом. Часовые пояса выражаются в виде смещений от UTC; например, в Нью-Йорке время отстает от UTC на 4 часа в летний период и на 5 часов в остальное время года.

В Python информация о часовых поясах берется из сторонней библиотеки `pytz`, которая является оберткой вокруг *базы данных Олсона*, где собраны все сведения о мировых часовых поясах. Это особенно важно для исторических данных, потому что даты перехода на летнее время (и даже смещения от UTC) многократно менялись по прихоти местных правительств. В США даты перехода на летнее время с 1900 года менялись много раз!

Подробные сведения о библиотеке `pytz` можно найти в документации к ней. Но поскольку pandas инкапсулирует функциональность `pytz`, то можете спокойно игнорировать весь ее API, кроме названий часовых поясов. А эти названия можно узнать как интерактивно, так и из документации:

```
In [418]: import pytz

In [419]: pytz.common_timezones[-5:]
Out[419]: ['US/Eastern', 'US/Hawaii', 'US/Mountain', 'US/Pacific', 'UTC']
```

Чтобы получить объект часового пояса от `pytz`, используется функция `pytz.timezone`:

```
In [420]: tz = pytz.timezone('US/Eastern')

In [421]: tz
Out[421]: <DstTzInfo 'US/Eastern' EST-1 day, 19:00:00 STD>
```

Методы из библиотеки pandas принимают как названия часовых зон, так и эти объекты. Я рекомендую использовать названия.

Локализация и преобразование

По умолчанию временные ряды в pandas не учитывают часовые пояса. Рассмотрим следующий ряд:

```
rng = pd.date_range('3/9/2012 9:30', periods=6, freq='D')
ts = Series(np.random.randn(len(rng)), index=rng)
```

Поле `tz` в индексе равно `None`:

```
In [423]: print(ts.index.tz)
None
```

Но при генерировании диапазонов дат можно и указать часовой пояс:

```
In [424]: pd.date_range('3/9/2012 9:30', periods=10, freq='D', tz='UTC')
Out[424]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-18 09:30:00]
Length: 10, Freq: D, Timezone: UTC
```

Для преобразования даты из инвариантного формата в *локализованный* служит метод `tz_localize`:

```
In [425]: ts_utc = ts.tz_localize('UTC')
In [426]: ts_utc
Out[426]:
2012-03-09 09:30:00+00:00     0.414615
2012-03-10 09:30:00+00:00     0.427185
2012-03-11 09:30:00+00:00     1.172557
2012-03-12 09:30:00+00:00    -0.351572
2012-03-13 09:30:00+00:00     1.454593
2012-03-14 09:30:00+00:00     2.043319
Freq: D

In [427]: ts_utc.index
Out[427]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]
Length: 6, Freq: D, Timezone: UTC
```

После локализации временного ряда для его преобразования в другой часовой пояс нужно вызвать метод `tz_convert`:

```
In [428]: ts_utc.tz_convert('US/Eastern')
Out[428]:
2012-03-09 04:30:00-05:00     0.414615
2012-03-10 04:30:00-05:00     0.427185
2012-03-11 05:30:00-04:00     1.172557
2012-03-12 05:30:00-04:00    -0.351572
2012-03-13 05:30:00-04:00     1.454593
```

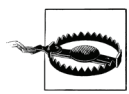
```
2012-03-14 05:30:00-04:00     2.043319
Freq: D
```

Приведенный выше временной ряд охватывает дату перехода на летнее время в восточном часовом поясе США, мы могли бы локализовать его для часового пояса, а затем преобразовать, скажем, в UTC или в берлинское время:

```
In [429]: ts_eastern = ts.tz_localize('US/Eastern')

In [430]: ts_eastern.tz_convert('UTC')
Out[430]:
2012-03-09 14:30:00+00:00     0.414615
2012-03-10 14:30:00+00:00     0.427185
2012-03-11 13:30:00+00:00     1.172557
2012-03-12 13:30:00+00:00    -0.351572
2012-03-13 13:30:00+00:00     1.454593
2012-03-14 13:30:00+00:00     2.043319
Freq: D

In [431]: ts_eastern.tz_convert('Europe/Berlin')
Out[431]:
2012-03-09 15:30:00+01:00     0.414615
2012-03-10 15:30:00+01:00     0.427185
2012-03-11 14:30:00+01:00     1.172557
2012-03-12 14:30:00+01:00    -0.351572
2012-03-13 14:30:00+01:00     1.454593
2012-03-14 14:30:00+01:00     2.043319
Freq: D
```

У объекта DatetimeIndex также существуют методы `tz_localize` и `tz_convert`:

```
In [432]: ts.index.tz_localize('Asia/Shanghai')
Out[432]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-09 09:30:00, ..., 2012-03-14 09:30:00]
Length: 6, Freq: D, Timezone: Asia/Shanghai
```



При локализации наивных временных меток проверяется также однозначность и существование моментов времени в окрестности даты перехода на летнее время.

Операции над объектами Timestamp с учетом часового пояса

По аналогии с временными рядами и диапазонами дат можно локализовать и отдельные объекты Timestamp, включив в них информацию о часовом поясе, а затем преобразовывать из одного пояса в другой:

```
In [433]: stamp = pd.Timestamp('2011-03-12 04:00')

In [434]: stamp_utc = stamp.tz_localize('utc')

In [435]: stamp_utc.tz_convert('US/Eastern')
Out[435]: <Timestamp: 2011-03-11 23:00:00-0500 EST, tz=US/Eastern>
```

Часовой пояс можно задать и при создании объекта Timestamp:

```
In [436]: stamp_moscow = pd.Timestamp('2011-03-12 04:00', tz='Europe/Moscow')

In [437]: stamp_moscow
Out[437]: <Timestamp: 2011-03-12 04:00:00+0300 MSK, tz=Europe/Moscow>
```

В объектах Timestamp, учитывающих часовой пояс, хранится временной штамп UTC в виде числа секунд от «эпохи» UNIX (1 января 1970); это значение инвариантно относительно преобразования из одного пояса в другой:

```
In [438]: stamp_utc.value
Out[438]: 1299902400000000000

In [439]: stamp_utc.tz_convert('US/Eastern').value
Out[439]: 1299902400000000000
```

При выполнении арифметических операций над объектами pandas `DateOffset` всюду, где возможно, учитывается переход на летнее время:

```
# за 30 минут до перехода на летнее время
In [440]: from pandas.tseries.offsets import Hour

In [441]: stamp = pd.Timestamp('2012-03-12 01:30', tz='US/Eastern')

In [442]: stamp
Out[442]: <Timestamp: 2012-03-12 01:30:00-0400 EDT, tz=US/Eastern>

In [443]: stamp + Hour()
Out[443]: <Timestamp: 2012-03-12 02:30:00-0400 EDT, tz=US/Eastern>

# за 90 минут до перехода на летнее время
In [444]: stamp = pd.Timestamp('2012-11-04 00:30', tz='US/Eastern')

In [445]: stamp
Out[445]: <Timestamp: 2012-11-04 00:30:00-0400 EDT, tz=US/Eastern>

In [446]: stamp + 2 * Hour()
Out[446]: <Timestamp: 2012-11-04 01:30:00-0500 EST, tz=US/Eastern>
```

Операции между датами из разных часовых поясов

Если комбинируются два временных ряда с разными часовыми поясами, то в результате получится UTC. Поскольку во внутреннем представлении временные метки хранятся в UTC, то операция не требует никаких преобразований:

```
In [447]: rng = pd.date_range('3/7/2012 9:30', periods=10, freq='B')

In [448]: ts = Series(np.random.randn(len(rng)), index=rng)

In [449]: ts
Out[449]:
```

```
2012-03-07 09:30:00   -1.749309
2012-03-08 09:30:00   -0.387235
2012-03-09 09:30:00   -0.208074
2012-03-12 09:30:00   -1.221957
2012-03-13 09:30:00   -0.067460
2012-03-14 09:30:00    0.229005
2012-03-15 09:30:00   -0.576234
2012-03-16 09:30:00    0.816895
2012-03-19 09:30:00   -0.772192
2012-03-20 09:30:00   -1.333576
Freq: B

In [450]: ts1 = ts[:7].tz_localize('Europe/London')

In [451]: ts2 = ts1[2:].tz_convert('Europe/Moscow')

In [452]: result = ts1 + ts2

In [453]: result.index
Out[453]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-07 09:30:00, ..., 2012-03-15 09:30:00]
Length: 7, Freq: B, Timezone: UTC
```

Периоды и арифметика периодов

Периоды – это промежутки времени: дни, месяцы, кварталы, годы. Этот тип данных представлен классом `Period`, конструктор которого принимает строку или число и частоту из приведенной выше таблицы:

```
In [454]: p = pd.Period(2007, freq='A-DEC')

In [455]: p
Out[455]: Period('2007', 'A-DEC')
```

В данном случае объект `Period` представляет промежуток времени от 1 января 2007 до 31 декабря 2007 включительно. Сложение и вычитание периода и целого числа дает тот же результат, что сдвиг на величину, кратную частоте периода:

```
In [456]: p + 5                          In [457]: p - 2
Out[456]: Period('2012', 'A-DEC')        Out[457]: Period('2005', 'A-DEC')
```

Если у двух периодов одинаковая частота, то их разностью является количество единиц частоты между ними:

```
In [458]: pd.Period('2014', freq='A-DEC') - p
Out[458]: 7
```

Регулярные диапазоны периодов строятся с помощью функции `period_range`:

```
In [459]: rng = pd.period_range('1/1/2000', '6/30/2000', freq='M')

In [460]: rng
```

```
Out[460]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2000-01, ..., 2000-06]
length: 6
```

В классе `PeriodIndex` хранится последовательность периодов, он может служить осевым индексом в любой структуре данных pandas:

```
In [461]: Series(np.random.randn(6), index=rng)
Out[461]:
2000-01   -0.309119
2000-02    0.028558
2000-03    1.129605
2000-04   -0.374173
2000-05   -0.011401
2000-06    0.272924
Freq: M
```

Если имеется массив строк, то можно обратиться к самому классу `PeriodIndex`:

```
In [462]: values = ['2001Q3', '2002Q2', '2003Q1']

In [463]: index = pd.PeriodIndex(values, freq='Q-DEC')

In [464]: index
Out[464]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: Q-DEC
[2001Q3, ..., 2003Q1]
length: 3
```

Преобразование частоты периода

Периоды и объекты `PeriodIndex` можно преобразовать с изменением частоты, воспользовавшись методом `asfreq`. Для примера предположим, что имеется годовой период, который мы хотим преобразовать в месячный, начинающийся или заканчивающийся на границе года. Это довольно просто:

```
In [465]: p = pd.Period('2007', freq='A-DEC')

In [466]: p.asfreq('M', how='start')    In [467]: p.asfreq('M', how='end')
Out[466]: Period('2007-01', 'M')        Out[467]: Period('2007-12', 'M')
```

Можно рассматривать `Period('2007', 'A-DEC')` как курсор, указывающий на промежуток времени, поделенный на периоды продолжительностью один месяц. Это проиллюстрировано на рис. 10.1. Для *финансового года*, заканчивающегося в любом месяце, кроме декабря, месячные подпериоды вычисляются по-другому:

```
In [468]: p = pd.Period('2007', freq='A-JUN')

In [469]: p.asfreq('M', 'start')     In [470]: p.asfreq('M', 'end')
Out[469]: Period('2006-07', 'M')     Out[470]: Period('2007-07', 'M')
```

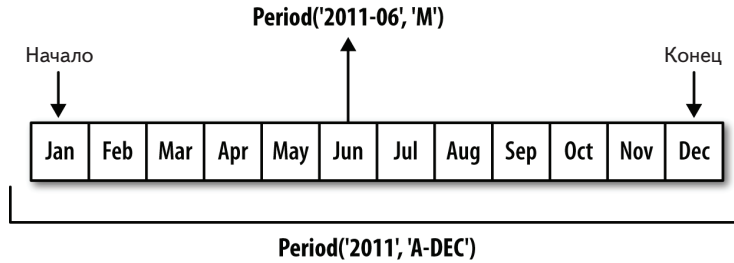


Рис. 10.1. Иллюстрация на тему преобразования частоты периода

Когда производится преобразование из большей частоты в меньшую, объемлющий период определяется в зависимости от того, куда попадает подпериод. Например, если частота равна `A-JUN`, то месяц `Aug-2007` фактически является частью периода `2008`:

```
In [471]: p = pd.Period('2007-08', 'M')

In [472]: p.asfreq('A-JUN')
Out[472]: Period('2008', 'A-JUN')
```

Эта семантика сохраняется и в случае преобразования целых объектов `PeriodIndex` или TimeSeries:

```
In [473]: rng = pd.period_range('2006', '2009', freq='A-DEC')

In [474]: ts = Series(np.random.randn(len(rng)), index=rng)

In [475]: ts
Out[475]:
2006   -0.601544
2007    0.574265
2008   -0.194115
2009    0.202225
Freq: A-DEC

In [476]: ts.asfreq('M', how='start')
Out[476]:
2006-01   -0.601544
2007-01    0.574265
2008-01   -0.194115
2009-01    0.202225
Freq: M

In [477]: ts.asfreq('B', how='end')
Out[477]:
2006-12-29   -0.601544
2007-12-31    0.574265
2008-12-31   -0.194115
2009-12-31    0.202225
Freq: B
```

Квартальная частота периода

Квартальные данные стандартно применяются в бухгалтерии, финансах и других областях. Обычно квартальные итоги подводятся относительно *конца финансового года*, каковым считается последний календарный или рабочий день одного из 12 месяцев. Следовательно, период `2012Q4` интерпретируется по-разному

в зависимости от того, что понимать под концом финансового года. Библиотека pandas поддерживает все 12 возможных значений квартальной частоты от `Q-JAN` до `Q-DEC`:

```
In [478]: p = pd.Period('2012Q4', freq='Q-JAN')

In [479]: p
Out[479]: Period('2012Q4', 'Q-JAN')
```

Если финансовый год заканчивается в январе, то период `2012Q4` охватывает месяцы с ноября по январь, и это легко проверить, преобразовав квартальную частоту в дневную. См. иллюстрацию на рис. 10.2.

```
In [480]: p.asfreq('D', 'start') In [481]: p.asfreq('D', 'end')
Out[480]: Period('2011-11-01', 'D') Out[481]: Period('2012-01-31', 'D')
```

Рис. 10.2. Различные соглашения о квартальной частоте

Таким образом, арифметические операции с периодами выполняются очень просто; например, чтобы получить временную метку для момента «4 часа пополудни предпоследнего рабочего дня квартала», нужно написать:

```
In [482]: p4pm = (p.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [483]: p4pm
Out[483]: Period('2012-01-30 16:00', 'T')

In [484]: p4pm.to_timestamp()
Out[484]: <Timestamp: 2012-01-30 16:00:00>
```

Генерация квартальных диапазонов работает ровно так, как естественно ожидать, – посредством `period_range`. Арифметические операции тоже аналогичны:

```
In [485]: rng = pd.period_range('2011Q3', '2012Q4', freq='Q-JAN')

In [486]: ts = Series(np.arange(len(rng)), index=rng)

In [487]: ts
Out[487]:
2011Q3   0
2011Q4   1
2012Q1   2
2012Q2   3
```

```
2012Q3   4
2012Q4   5
Freq: Q-JAN

In [488]: new_rng = (rng.asfreq('B', 'e') - 1).asfreq('T', 's') + 16 * 60

In [489]: ts.index = new_rng.to_timestamp()

In [490]: ts
Out[490]:
2010-10-28 16:00:00   0
2011-01-28 16:00:00   1
2011-04-28 16:00:00   2
2011-07-28 16:00:00   3
2011-10-28 16:00:00   4
2012-01-30 16:00:00   5
```

Преобразование временных меток в периоды и обратно

Объекты Series и DataFrame, индексированные временными метками, можно преобразовать в периоды методом `to_period`:

```
In [491]: rng = pd.date_range('1/1/2000', periods=3, freq='M')

In [492]: ts = Series(randn(3), index=rng)

In [493]: pts = ts.to_period()

In [494]: ts                  In [495]: pts
Out[494]:                     Out[495]:
2000-01-31  -0.505124         2000-01  -0.505124
2000-02-29   2.954439         2000-02   2.954439
2000-03-31  -2.630247         2000-03  -2.630247
Freq: M Freq: M
```

Поскольку периоды всегда ссылаются на непересекающиеся временные промежутки, то временная метка может принадлежать только одному периоду данной частоты. Можно задать любую частоту, а частота нового объекта `PeriodIndex` по умолчанию выводится из временных меток. Наличие повторяющихся периодов в результате также не приводит ни к каким проблемам:

```
In [496]: rng = pd.date_range('1/29/2000', periods=6, freq='D')

In [497]: ts2 = Series(randn(6), index=rng)

In [498]: ts2.to_period('M')
Out[498]:
2000-01  -0.352453
2000-01  -0.477808
2000-01   0.161594
2000-02   1.686833
```

```
2000-02   0.821965
2000-02  -0.667406
Freq: M
```

Для обратного преобразования во временные метки служит метод `to_timestamp`:

```
In [499]: pts = ts.to_period()

In [500]: pts
Out[500]:
2000-01  -0.505124
2000-02   2.954439
2000-03   -2.630247
Freq: M

In [501]: pts.to_timestamp(how='end')
Out[501]:
2000-01-31  -0.505124
2000-02-29   2.954439
2000-03-31   -2.630247
Freq: M
```

Создание PeriodIndex из массивов

В наборах данных с фиксированной частотой информация о промежутках времени иногда хранится в нескольких столбцах. Например, в следующем макроэкономическом наборе данных год и квартал находятся в разных столбцах:

```
In [502]: data = pd.read_csv('ch08/macrodata.csv')

In [503]: data.year       In [504]: data.quarter
Out[503]:                 Out[504]:
0    1959                 0    1
1    1959                 1    2
2    1959                 2    3
3    1959                 3    4
...                       ...
199  2008                 199  4
200  2009                 200  1
201  2009                 201  2
202  2009                 202  3
Name: year, Length: 203 Name: quarter, Length: 203
```

Передав эти массивы конструктору `PeriodIndex` вместе с частотой, мы сможем объединить их для построения индекса DataFrame:

```
In [505]: index = pd.PeriodIndex(year=data.year, quarter=data.quarter,
   .....                          freq='Q-DEC')

In [506]: index
Out[506]:
<class 'pandas.tseries.period.PeriodIndex'>
```

```
freq: Q-DEC
[1959Q1, ..., 2009Q3]
length: 203

In [507]: data.index = index

In [508]: data.infl
Out[508]:
1959Q1   0.00
1959Q2   2.34
1959Q3   2.74
1959Q4   0.27
...
2008Q4  -8.79
2009Q1   0.94
2009Q2   3.37
2009Q3   3.56
Freq: Q-DEC, Name: infl, Length: 203
```

Передискретизация и преобразование частоты

Под *передискретизацией* понимается процесс изменения частоты временного ряда. Агрегирование с переходом от высокой частоты к низкой называется *понижающей передискретизацией*, а переход от низкой частоты к более высокой – *повышающей передискретизацией*. Не всякая передискретизация попадает в одну из этих категорий; например, преобразование частоты `W-WED` (еженедельно по средам) в `W-FRI` не повышает и не понижает частоту.

Все объекты pandas имеют метод `resample`, отвечающий за любые преобразования частоты:

```
In [509]: rng = pd.date_range('1/1/2000', periods=100, freq='D')

In [510]: ts = Series(randn(len(rng)), index=rng)

In [511]: ts.resample('M', how='mean')
Out[511]:
2000-01-31   0.170876
2000-02-29   0.165020
2000-03-31   0.095451
2000-04-30   0.363566
Freq: M

In [512]: ts.resample('M', how='mean', kind='period')
Out[512]:
2000-01   0.170876
2000-02   0.165020
2000-03   0.095451
2000-04   0.363566
Freq: M
```

Метод `resample` обладает большой гибкостью и работает быстро, так что применим даже к очень большим временным рядам. Я проиллюстрирую его семантику и использование на нескольких примерах.

Таблица 10.5. Аргументы метода `resample`

Аргумент	Описание
`freq`	Строка или объект DateOffset, задающий новую частоту, например: `'M'`, `'5min'` или `Second(15)`
`how='mean'`	Имя функции или функция, порождающая агрегированное значение, например `'mean'`. По умолчанию `'mean'`. Допустимы также значения `'first'`, `'last'`, `'median'`, `'ohlc'`, `'max'`, `'min'`
`axis=0`	Ось передискретизации, по умолчанию 0
`fill_method=None`	Способ интерполяции при повышающей передискретизации, например `'ffill'` или `'bfill'`. По умолчанию интерполяция не производится.
`closed='right'`	При понижающей передискретизации определяет, какой конец интервала должен включаться: `'right'` (правый) или `'left'` (левый)
`label='right'`	При понижающей передискретизации определяет, следует ли помечать агрегированный результат меткой правого или левого конца интервала. Например, пятиминутный интервал от 9:30 to 9:35 можно пометить меткой 9:30 или 9:35. По умолчанию `'right'` (т. е. 9:35 в этом примере)
`loffset=None`	Поправка для времени меток интервалов, например `'-1s'` / `Second(-1)`, чтобы сдвинуть метки агрегатов на одну секунду назад
`limit=None`	При прямом или обратном восполнении максимальное количество восполняемых периодов
`kind=None`	Агрегировать в периоды (`'period'`) или временные метки (`'timestamp'`); по умолчанию определяется видом индекса, связанного с данным временным рядом
`convention='end'`	При передискретизации периодов соглашение (`'start'` или `'end'`) о преобразовании периода низкой частоты в период высокой частоты. По умолчанию `'end'`

Понижающая передискретизация

Агрегирование данных с целью понижения и регуляризации частоты – задача, которая часто встречается при работе с временными рядами. Частота определяет границы интервалов, разбивающих агрегируемые данные на порции. Например, для преобразования к месячному периоду `'M'` или `'BM'` данные нужно разбить на интервалы продолжительностью один месяц. Говорят, что каждый интервал *полуоткрыт*; любая точка может принадлежать только одному интервалу, а их объединение должно покрывать всю протяженность временного ряда. Перед тем как выполнять понижающую передискретизацию данных методом `resample`, нужно решить для себя следующие вопросы:

- какой конец интервала будет включаться;
- помечать ли агрегированный интервал меткой его начала или конца.

Для иллюстрации рассмотрим данные с частотой одна минута:

```
In [513]: rng = pd.date_range('1/1/2000', periods=12, freq='T')

In [514]: ts = Series(np.arange(12), index=rng)

In [515]: ts
Out[515]:
2000-01-01 00:00:00   0
2000-01-01 00:01:00   1
2000-01-01 00:02:00   2
2000-01-01 00:03:00   3
2000-01-01 00:04:00   4
2000-01-01 00:05:00   5
2000-01-01 00:06:00   6
2000-01-01 00:07:00   7
2000-01-01 00:08:00   8
2000-01-01 00:09:00   9
2000-01-01 00:10:00  10
2000-01-01 00:11:00  11
Freq: T
```

Пусть требуется агрегировать данные в пятиминутные группы, или *столбики*, вычислив сумму по каждой группе:

```
In [516]: ts.resample('5min', how='sum')
Out[516]:
2000-01-01 00:00:00   0
2000-01-01 00:05:00  15
2000-01-01 00:10:00  40
2000-01-01 00:15:00  11
Freq: 5T
```

Переданная частота определяет границы интервалов с пятиминутным приращением. По умолчанию включается *правый* конец интервала, т. е. значение `00:05` включается в интервал от `00:00` до `00:05`[1]. Если задать параметр `closed='left'`, то будет включаться левый конец интервала:

```
In [517]: ts.resample('5min', how='sum', closed='left')
Out[517]:
2000-01-01 00:05:00   10
2000-01-01 00:10:00   35
2000-01-01 00:15:00   21
Freq: 5T
```

Как видите, результирующий временной ряд помечен временными метками, соответствующими правым концам интервалов. Параметр `label='left'` позволяет использовать для этой цели метки левых концов:

---

[1] Выбор значений `closed='right'`, `label='right'` по умолчанию некоторым пользователям может показаться странным. На практике выбор более-менее произволен, для одних частот более естественным выглядит `closed='right'`, для других `closed='left'`. Важно лишь не забывать, как именно сегментированы данные.

```
In [518]: ts.resample('5min', how='sum', closed='left', label='left')
Out[518]:
2000-01-01 00:00:00  10
2000-01-01 00:05:00  35
2000-01-01 00:10:00  21
Freq: 5T
```

На рис. 10.3 показано, как данные с минутной частотой агрегируются в пятиминутные группы.

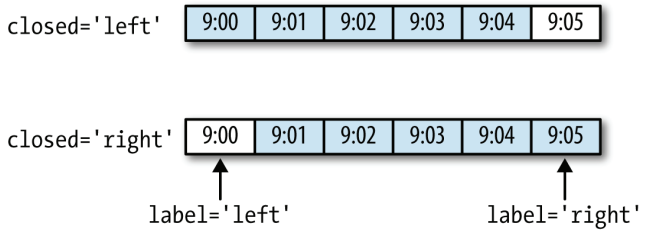


Рис. 10.3. Соглашения о включении конца и о метках интервалов на примере передискретизации с частотой 5 минут

Наконец, иногда желательно сдвинуть индекс результата на какую-то величину, скажем, вычесть одну секунду из правого конца, чтобы было понятнее, к какому интервалу относится временная метка. Для этого следует задать строку или смещение даты в параметре `loffset`:

```
In [519]: ts.resample('5min', how='sum', loffset='-1s')
Out[519]:
1999-12-31 23:59:59   0
2000-01-01 00:04:59  15
2000-01-01 00:09:59  40
2000-01-01 00:14:59  11
Freq: 5T
```

Того же результата можно достичь, вызвав метод `shift` объекта `ts` без параметра `loffset`.

Передискретизация OHLC

В финансовых приложениях очень часто временной ряд агрегируют, вычисляя четыре значения для каждого интервала: первое (открытие – open), последнее (закрытие – close), максимальное (high) и минимальное (low). Задав параметр `how='ohlc'`, мы получим объект DataFrame, в столбцах которого находятся эти четыре агрегата, которые эффективно вычисляются за один проход:

```
In [520]: ts.resample('5min', how='ohlc')
Out[520]:
                     open  high  low  close
2000-01-01 00:00:00     0     0    0      0
2000-01-01 00:05:00     1     5    1      5
2000-01-01 00:10:00     6    10    6     10
2000-01-01 00:15:00    11    11   11     11
```

Передискретизация с помощью GroupBy

Альтернативный способ понижающей передискретизации состоит в том, чтобы воспользоваться механизмом pandas `groupby`. Например, можно сгруппировать данные по месяцам или по рабочим дням, передав функцию, которая обращается к соответствующим поля индекса временного ряда:

```
In [521]: rng = pd.date_range('1/1/2000', periods=100, freq='D')

In [522]: ts = Series(np.arange(100), index=rng)

In [523]: ts.groupby(lambda x: x.month).mean()
Out[523]:
1    15
2    45
3    75
4    95

In [524]: ts.groupby(lambda x: x.weekday).mean()
Out[524]:
0    47.5
1    48.5
2    49.5
3    50.5
4    51.5
5    49.0
6    50.0
```

Повышающая передискретизация и интерполяция

Для преобразования от низкой частоты к более высокой агрегирование не требуется. Рассмотрим объект DataFrame, содержащий недельные данные:

```
In [525]: frame = DataFrame(np.random.randn(2, 4),
   .....:                   index=pd.date_range('1/1/2000', periods=2, freq='W-WED'),
   .....:                       columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [526]: frame[:5]
Out[526]:
             Colorado      Texas   New York      Ohio
2000-01-05  -0.609657  -0.268837   0.195592   0.85979
2000-01-12  -0.263206   1.141350  -0.101937  -0.07666
```

После передискретизации на частоту в один день по умолчанию вставляются отсутствующие значения:

```
In [527]: df_daily = frame.resample('D')

In [528]: df_daily
Out[528]:
             Colorado      Texas   New York      Ohio
2000-01-05  -0.609657  -0.268837   0.195592   0.85979
```

```
2000-01-06        NaN        NaN        NaN       NaN
2000-01-07        NaN        NaN        NaN       NaN
2000-01-08        NaN        NaN        NaN       NaN
2000-01-09        NaN        NaN        NaN       NaN
2000-01-10        NaN        NaN        NaN       NaN
2000-01-11        NaN        NaN        NaN       NaN
2000-01-12  -0.263206   1.141350  -0.101937  -0.07666
```

Допустим, мы хотим восполнить значения для дней, отличных от среды. Для этого применимы те же способы восполнения или интерполяции, что в методах `fillna` и `reindex`:

```
In [529]: frame.resample('D', fill_method='ffill')
Out[529]:
             Colorado      Texas   New York      Ohio
2000-01-05  -0.609657  -0.268837   0.195592   0.85979
2000-01-06  -0.609657  -0.268837   0.195592   0.85979
2000-01-07  -0.609657  -0.268837   0.195592   0.85979
2000-01-08  -0.609657  -0.268837   0.195592   0.85979
2000-01-09  -0.609657  -0.268837   0.195592   0.85979
2000-01-10  -0.609657  -0.268837   0.195592   0.85979
2000-01-11  -0.609657  -0.268837   0.195592   0.85979
2000-01-12  -0.263206   1.141350  -0.101937  -0.07666
```

Можно восполнить отсутствующие значения не во всех последующих периодах, а только в заданном числе:

```
In [530]: frame.resample('D', fill_method='ffill', limit=2)
Out[530]:
             Colorado      Texas   New York      Ohio
2000-01-05  -0.609657  -0.268837   0.195592   0.85979
2000-01-06  -0.609657  -0.268837   0.195592   0.85979
2000-01-07  -0.609657  -0.268837   0.195592   0.85979
2000-01-08        NaN        NaN        NaN       NaN
2000-01-09        NaN        NaN        NaN       NaN
2000-01-10        NaN        NaN        NaN       NaN
2000-01-11        NaN        NaN        NaN       NaN
2000-01-12  -0.263206   1.141350  -0.101937  -0.07666
```

Важно отметить, что новый индекс дат может вообще не пересекаться со старым:

```
In [531]: frame.resample('W-THU', fill_method='ffill')

Out[531]:
Colorado Texas New York Ohio
2000-01-06  -0.609657  -0.268837   0.195592   0.85979
2000-01-13  -0.263206   1.141350  -0.101937  -0.07666
```

Передискретизация периодов

Передискретизация данных, индексированных периодами, производится достаточно просто и работает в соответствии с ожиданиями:

```
In [532]: frame = DataFrame(np.random.randn(24, 4),
   .....:                   index=pd.period_range('1-2000', '12-2001', freq='M'),
   .....:                   columns=['Colorado', 'Texas', 'New York', 'Ohio'])

In [533]: frame[:5]
Out[533]:
            Colorado      Texas   New York       Ohio
2000-01   0.120837   1.076607   0.434200   0.056432
2000-02  -0.378890   0.047831   0.341626   1.567920
2000-03  -0.047619  -0.821825  -0.179330  -0.166675
2000-04   0.333219  -0.544615  -0.653635  -2.311026
2000-05   1.612270  -0.806614   0.557884   0.580201

In [534]: annual_frame = frame.resample('A-DEC', how='mean')

In [535]: annual_frame
Out[535]:
       Colorado      Texas   New York       Ohio
2000   0.352070  -0.553642   0.196642  -0.094099
2001   0.158207   0.042967  -0.360755   0.184687
```

Повышающая передискретизация чуть сложнее, потому что необходимо принять решение о том, в какой конец промежутка времени для новой частоты помещать значения до передискретизации, как в случае метода `asfreq`. Аргумент `convention` по умолчанию равен `'end'`, но можно задать и значение `'start'`:

```
# Q-DEC: поквартально, год заканчивается в декабре
In [536]: annual_frame.resample('Q-DEC', fill_method='ffill')
Out[536]:
        Colorado      Texas   New York       Ohio
2000Q4  0.352070  -0.553642   0.196642  -0.094099
2001Q1  0.352070  -0.553642   0.196642  -0.094099
2001Q2  0.352070  -0.553642   0.196642  -0.094099
2001Q3  0.352070  -0.553642   0.196642  -0.094099
2001Q4  0.158207   0.042967  -0.360755   0.184687

In [537]: annual_frame.resample('Q-DEC', fill_method='ffill',
   .....                         convention='start')
Out[537]:
        Colorado      Texas   New York       Ohio
2000Q1  0.352070  -0.553642   0.196642  -0.094099
2000Q2  0.352070  -0.553642   0.196642  -0.094099
2000Q3  0.352070  -0.553642   0.196642  -0.094099
2000Q4  0.352070  -0.553642   0.196642  -0.094099
2001Q1  0.158207   0.042967  -0.360755   0.184687
```

Поскольку периоды ссылаются на промежутки времени, правила повышающей и понижающей передискретизации более строгие:

- в случае понижающей передискретизации конечная частота должна быть *подпериодом* начальной;
- в случае повышающей передискретизации конечная частота должна быть *надпериодом* начальной.

Если эти правила не выполнены, то будет возбуждено исключение. Это относится главным образом к квартальной, годовой и недельной частоте; например, промежутки времени, определенные частотой `Q-MAR`, выровнены только с периодами `A-MAR`, `A-JUN`, `A-SEP` и `A-DEC`:

```
In [538]: annual_frame.resample('Q-MAR', fill_method='ffill')
Out[538]:
        Colorado      Texas   New York       Ohio
2001Q3  0.352070  -0.553642   0.196642  -0.094099
2001Q4  0.352070  -0.553642   0.196642  -0.094099
2002Q1  0.352070  -0.553642   0.196642  -0.094099
2002Q2  0.352070  -0.553642   0.196642  -0.094099
2002Q3  0.158207   0.042967  -0.360755   0.184687
```

Графики временных рядов

В средствах построения графиков временных рядов в pandas улучшено форматирование дат по сравнению со штатным пакетом matplotlib. В качестве примера я скачал с сайта Yahoo! Finance данные о котировках акций нескольких известных американских компаний:

```
In [539]: close_px_all = pd.read_csv('ch09/stock_px.csv', parse_dates=True,
   .....                              index_col=0)

In [540]: close_px = close_px_all[['AAPL', 'MSFT', 'XOM']]

In [541]: close_px = close_px.resample('B', fill_method='ffill')

In [542]: close_px
Out[542]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2292 entries, 2003-01-02 00:00:00 to 2011-10-14 00:00:00
Freq: B
Data columns:
AAPL    2292  non-null values
MSFT    2292  non-null values
XOM     2292  non-null values
dtypes: float64(3)
```

Вызов метода `plot` для одного из столбцов строит график, показанный на рис. 10.4.

```
In [544]: close_px['AAPL'].plot()
```

При вызове от имени DataFrame все временные ряды, как и следовало ожидать, рисуются в одном подграфике, а в пояснительной надписи описывается, что есть что. Я построил график только для данных за 2009 год, чтобы было хорошо видно, как форматируются месяцы и годы на оси X (рис. 10.5).

```
In [546]: close_px.ix['2009'].plot()
In [548]: close_px['AAPL'].ix['01-2011':'03-2011'].plot()
```

Рис. 10.4. Суточные котировки AAPL

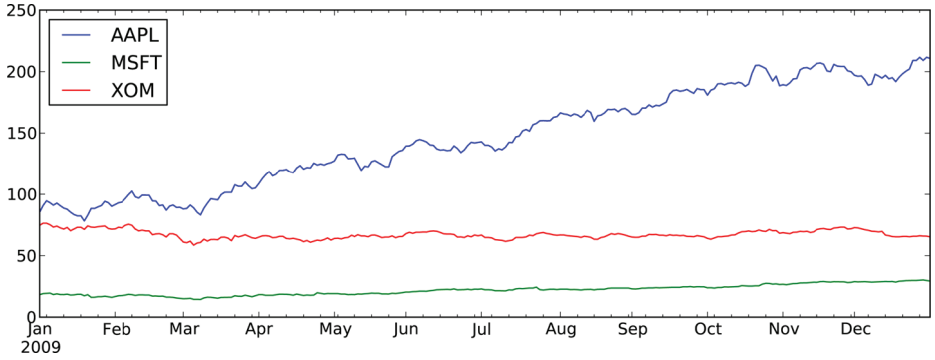


Рис. 10.5. Котировки акций в 2009 году

Рис. 10.6. Суточные котировки акций Apple за период 1/2011–3/2011

Данные с частотой один квартал также отформатированы более красиво: проставлены маркеры кварталов, что вручную делать довольно утомительно. См. рис. 10.7.

```
In [550]: appl_q = close_px['AAPL'].resample('Q-DEC', fill_method='ffill')
In [551]: appl_q.ix['2009':].plot()
```

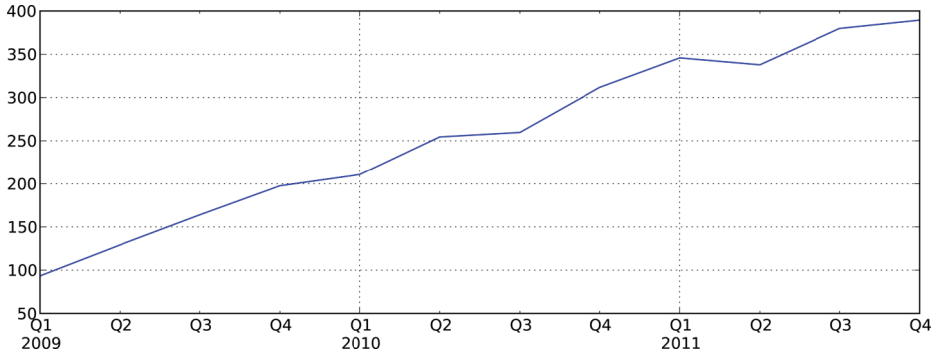


Рис. 10.7. Квартальные котировки акций Apple за период 2009–2011

И последняя особенность графиков временных рядов в pandas заключается в том, что при буксировке с нажатой правой кнопкой мыши данные динамически расширяются или сжимаются (с переформатированием) в зависимости от промежутка времени в представлении графика. Разумеется, для этого matplotlib должна использоваться в интерактивном режиме.

Скользящие оконные функции

Распространенный класс преобразований массива, применяемый для операций с временными рядами, – статистические и иные функции, вычисляемые в скользящем окне или с экспоненциально убывающими весами. Я называю их *скользящими оконными функциями*, хотя сюда относятся также функции, не связанные с окном постоянной ширины, например, экспоненциально взвешенное скользящее среднее. Как и во всех статистических функциях, отсутствующие значения автоматически отбрасываются.

Одной из простейших функций такого рода является `rolling_mean`. Она получает объект TimeSeries или DataFrame и окно `window` (заданное как число периодов):

```
In [555]: close_px.AAPL.plot()
Out[555]: <matplotlib.axes.AxesSubplot at 0x1099b3990>

In [556]: pd.rolling_mean(close_px.AAPL, 250).plot()
```

Ее график показан на рис. 10.8. По умолчанию функции типа `rolling_mean` требуют задания числа результатов измерения, отличных от NA. Это поведение можно изменить, так чтобы принимались во внимание отсутствующие значения и, прежде всего, тот факт, что в начале временного ряда данных меньше, чем `window` периодов (рис. 10.9).

```
In [558]: appl_std250 = pd.rolling_std(close_px.AAPL, 250, min_periods=10)

In [559]: appl_std250[5:12]
Out[559]:
```

```
2003-01-09       NaN
2003-01-10       NaN
2003-01-13       NaN
2003-01-14       NaN
2003-01-15  0.077496
2003-01-16  0.074760
2003-01-17  0.112368
Freq: B

In [560]: appl_std250.plot()
```

Рис. 10.8. Котировки акций Apple со скользящим средним за 250 дней

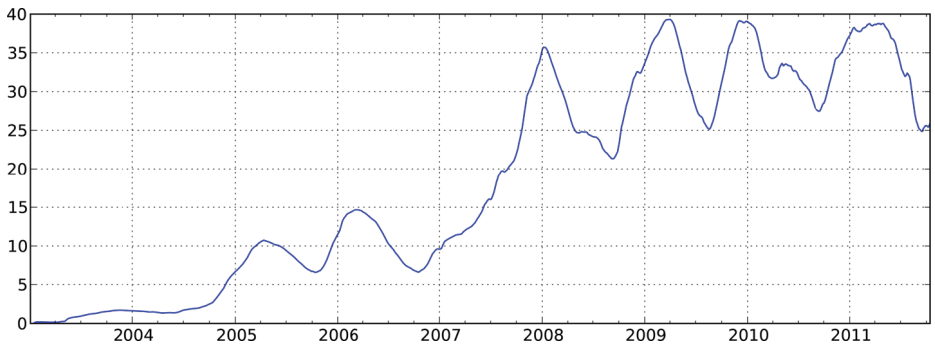


Рис. 10.9. Скользящее стандартное отклонение суточного оборота акций Apple за 250 дней

При вычислении *среднего с расширяющимся окном* видно, что расширяющееся окно – просто частный случай, когда длина окна совпадает с длиной временного ряда, а для вычисления значения требуется один или более периодов:

```
# Определяем среднее с расширяющимся окном в терминах rolling_mean
In [561]: expanding_mean = lambda x: rolling_mean(x, len(x), min_periods=1)
```

При вызове метода `rolling_mean` и родственных ему для объекта DataFrame преобразование применяется к каждому столбцу (рис. 10.10):

```
In [563]: pd.rolling_mean(close_px, 60).plot(logy=True)
```

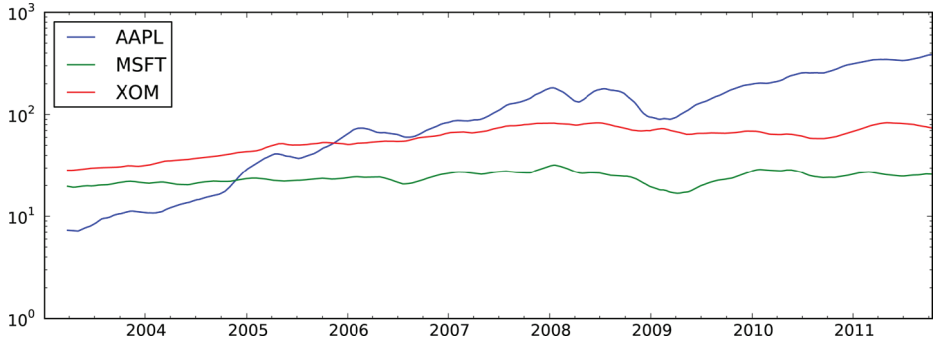


Рис. 10.10. Скользящее среднее котировок акций за 60 дней (с логарифмическим масштабом по оси Y)

В табл. 10.6 перечислены соответствующие функции в библиотеке pandas.

Таблица 10.6. Скользящие оконные функции и экспоненциально взвешенные функции

Функция	Описание
`rolling_count`	Возвращает количество отличных от NA результатов измерения в каждом скользящем окне
`rolling_sum`	Скользящая сумма
`rolling_mean`	Скользящее среднее
`rolling_median`	Скользящая медиана
`rolling_var, rolling_std`	Скользящая дисперсия и скользящее стандартное отклонение соответственно. В знаменателе $n - 1$
`rolling_skew, rolling_kurt`	Скользящая асимметрия (третий момент) и скользящий куртозис (четвертый момент) соответственно
`rolling_min, rolling_max`	Скользящий минимум и максимум
`rolling_quantile`	Скользящая оценка для заданного процентиля (выборочного квантиля)
`rolling_corr, rolling_cov`	Скользящая корреляция и ковариация
`rolling_apply`	Применяет обобщенную функцию к массиву значений в скользящем окне
`ewma`	Экспоненциально взвешенное скользящее среднее
`ewmvar, ewmstd`	Экспоненциально взвешенная скользящая дисперсия и стандартное отклонение
`ewmcorr, ewmcov`	Экспоненциально взвешенная скользящая корреляция и ковариация

Разработанная Китом Гудмэном (Keith Goodman) библиотека `bottleneck` содержит альтернативную реализацию скользящих оконных функций, учитывающих значения `NaN`. Для некоторых приложений она может оказаться более удобной.

Экспоненциально взвешенные функции

Вместо использования окна постоянного размера, когда веса всех наблюдений одинаковы, можно задать постоянный *коэффициент затухания*, чтобы повысить вес последних наблюдений. В математических обозначениях это формулируется так: если ma_t – скользящее среднее в момент t, а x – рассматриваемый временной ряд, то результирующие значения вычисляются по формуле

$$ma_t = a * ma_{t-1} + (a - 1) * x_{-t},$$

где a – коэффициент затухания. Есть два способа задать коэффициент затухания, самый популярный – использовать *промежуток* (span), потому что результаты в этом случае получаются сравнимыми с применением простой скользящей оконной функции, для которой размер окна равен промежутку.

Поскольку экспоненциально взвешенная статистика придает больший вес недавним наблюдениям, она быстрее «адаптируется» к изменениям по сравнению с вариантом с равными весами. Ниже приведен пример сравнения скользящего среднего котировок акций Apple за 60 дней с экспоненциально взвешенным скользящим средним для `span=60` (рис. 10.11):

```
fig, axes = plt.subplots(nrows=2, ncols=1, sharex=True, sharey=True,
                         figsize=(12, 7))

aapl_px = close_px.AAPL['2005':'2009']

ma60 = pd.rolling_mean(aapl_px, 60, min_periods=50)
ewma60 = pd.ewma(aapl_px, span=60)

aapl_px.plot(style='k-', ax=axes[0])
ma60.plot(style='k--', ax=axes[0])
aapl_px.plot(style='k-', ax=axes[1])
ewma60.plot(style='k--', ax=axes[1])
axes[0].set_title('Simple MA')
axes[1].set_title('Exponentially-weighted MA')
```

Бинарные скользящие оконные функции

Для некоторых статистических операций, в частности корреляции и ковариации, необходимы два временных ряда. Например, финансовых аналитиков часто интересует корреляция цены акции с основным биржевым индексом типа S&P 500. Чтобы найти эту величину, нужно вычислить относительные изменения в процентах и воспользоваться функцией `rolling_corr` (рис. 10.12):

```
In [570]: spx_rets = spx_px / spx_px.shift(1) - 1

In [571]: returns = close_px.pct_change()

In [572]: corr = pd.rolling_corr(returns.AAPL, spx_rets, 125, min_periods=100)

In [573]: corr.plot()
```

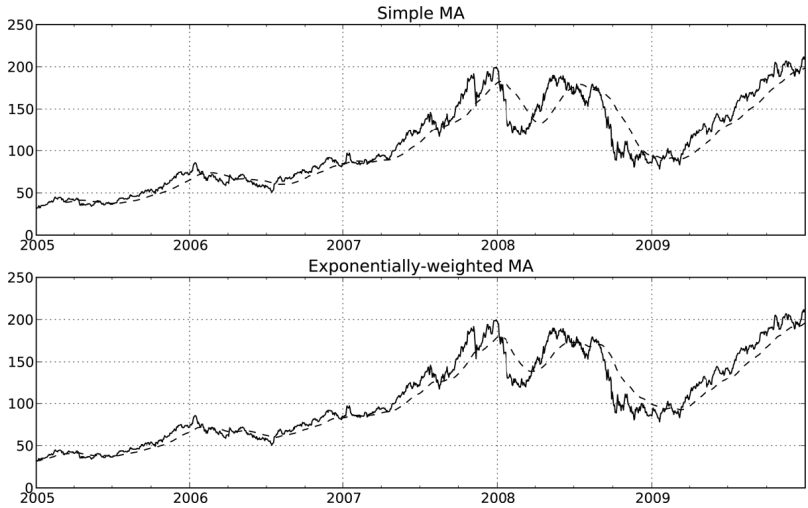


Рис. 10.11. Простое и экспоненциально взвешенное скользящее среднее

Рис. 10.12. Корреляция доходности AAPL с индексом S&P 500, рассчитанная за шесть месяцев

Пусть требуется вычислить корреляцию индекса S&P 500 сразу с несколькими акциями. Писать каждый раз цикл и создавать новый объект DataFrame, конечно, нетрудно, но уж больно скучно, поэтому если передать функции `rolling_corr` (или ей подобной) объекты TimeSeries и DataFrame, то она вычислит корреляцию TimeSeries (в данном случае `spx_rets`) с каждым столбцом DataFrame. Результат показан на рис. 10.13.

```
In [575]: corr = pd.rolling_corr(returns, spx_rets, 125, min_periods=100)

In [576]: corr.plot()
```

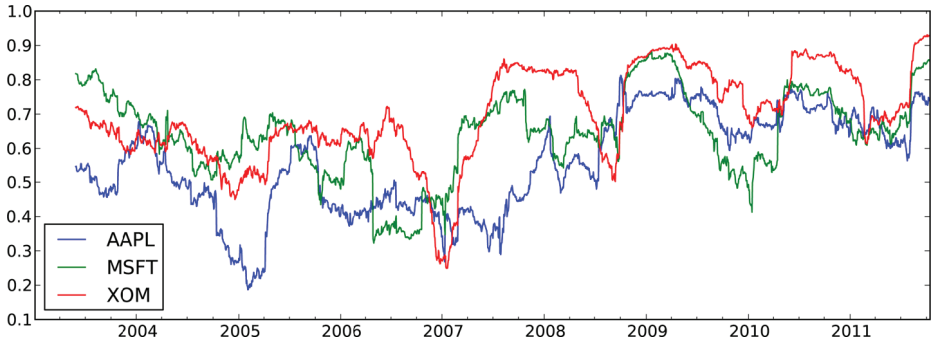


Рис. 10.13. Корреляция доходности нескольких акций с индексом S&P 500, рассчитанная за шесть месяцев

Рис. 10.14. Двухпроцентный процентильный ранг доходности AAPL, рассчитанный по окну протяженностью 1 год

Скользящие оконные функции, определенные пользователем

Функция `rolling_apply` позволяет применить произвольную функцию, принимающую массив, к скользящему окну. Единственное требование заключается в том, что функция должна порождать единственное скалярное значение (производить редукцию) для каждого фрагмента массива. Например, при вычислении выборочных квантилей с помощью `rolling_quantile` нам может быть интересен процентильный ранг некоторого значения относительно выборки. Это можно сделать с помощью функции `scipy.stats.percentileofscore`:

```
In [578]: from scipy.stats import percentileofscore

In [579]: score_at_2percent = lambda x: percentileofscore(x, 0.02)

In [580]: result = pd.rolling_apply(returns.AAPL, 250, score_at_2percent)

In [581]: result.plot()
```

Замечания о быстродействии и потреблении памяти

Временные метки и периоды представлены 64-разрядными целыми числами с помощью типа NumPy `datetime64`. Это означает, что для каждого наблюдения накладные расходы на хранение временной метки составляют 8 байт. Следовательно, временной ряд, состоящий из 1 миллиона наблюдений типа `float64`, займет в памяти примерно 16 мегабайт. Поскольку pandas делает все возможное для разделения индексов между временными рядами, при создании нового представления уже имеющегося временного ряда дополнительная память не выделяется. Кроме того, индексы с низкой частотой (от одного дня и более) хранятся в центральном кэше, так что любой индекс с фиксированной частотой является представлением кэша дат. Таким образом, при наличии большого числа временных рядов с низкой частотой расход памяти на хранение индексов будет не очень значительным.

В плане быстродействия pandas оптимизирована для операций выравнивания данных (которое производится подспудно, например, при сложении `ts1 + ts2` по-разному проиндексированных временных рядов) и передискретизации. Вот пример OHLC-агрегирования 10 миллионов наблюдений:

```
In [582]: rng = pd.date_range('1/1/2000', periods=10000000, freq='10ms')

In [583]: ts = Series(np.random.randn(len(rng)), index=rng)

In [584]: ts
Out[584]:
2000-01-01 00:00:00         -1.402235
2000-01-01 00:00:00.010000   2.424667
2000-01-01 00:00:00.020000  -1.956042
2000-01-01 00:00:00.030000  -0.897339
...
2000-01-02 03:46:39.960000   0.495530
2000-01-02 03:46:39.970000   0.574766
2000-01-02 03:46:39.980000   1.348374
2000-01-02 03:46:39.990000   0.665034
Freq: 10L, Length: 10000000

In [585]: ts.resample('15min', how='ohlc')
Out[585]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 113 entries, 2000-01-01 00:00:00 to 2000-01-02 04:00:00
Freq: 15T
Data columns:
open    113  non-null values
high    113  non-null values
low     113  non-null values
close   113  non-null values
dtypes: float64(4)

In [586]: %timeit ts.resample('15min', how='ohlc')
10 loops, best of 3: 61.1 ms per loop
```

Время работы может слабо зависеть от относительного размера результата агрегирования. Чем выше частота, тем больше время вычислений, что и неудивительно:

```
In [587]: rng = pd.date_range('1/1/2000', periods=10000000, freq='1s')

In [588]: ts = Series(np.random.randn(len(rng)), index=rng)

In [589]: %timeit ts.resample('15s', how='ohlc')
1 loops, best of 3: 88.2 ms per loop
```

Не исключено, что к моменту, когда вы будете читать этот текст, производительность алгоритмов еще возрастет. Например, сейчас преобразование между регулярными частотами не оптимизировано, но это нетрудно было бы сделать.

ГЛАВА 11. Финансовые и экономические приложения

Начиная с 2005 года, быстро растет число применений языка Python в финансовых приложениях, что обусловлено, прежде всего, появлением зрелых библиотек (типа NumPy и pandas) и доступностью пишущих на этом языке программистов. Различные организации пришли к выводу, что Python отлично приспособлен как для интерактивного анализа, так и для разработки надежных систем в гораздо более сжатые сроки, чем было бы возможно на Java или C++. К тому же, Python – идеальный связующий слой; нетрудно написать интерфейсы из Python к унаследованным библиотекам на C или C++.

Хотя финансовый анализ – область настолько обширная, что ей можно было бы посвятить целую книгу, я надеюсь все же продемонстрировать, как представленные в этой книге средства можно применить к различным конкретным задачам. Как и во всех приложениях, связанных с анализом данных, программист часто тратит больше усилий на переформатирование данных, чем на само моделирование и исследование. Лично я начал писать библиотеку pandas в 2008 году, потому что устал бороться с неподходящими инструментами.

В примерах ниже я буду использовать термин *срез* для обозначения данных в конкретный момент времени. Например, цены всех акций в момент закрытия торгов в биржевом индексе S&P 500 на конкретную дату образуют срез. Срезы различных данных в разные моменты времени (например, цены и объемы торгов) образуют *панель*. Панель данных можно представить в виде объекта DataFrame с иерархическим индексом или в виде трехмерного объекта Panel из библиотеки pandas.

О переформатировании данных

В предыдущих главах встречалось немало средств переформатирования, полезных в финансовых приложениях. Здесь я остановлюсь на нескольких вопросах, имеющих прямое отношение к рассматриваемой предметной области.

Временные ряды и выравнивание срезов

При работе с финансовыми данными очень много времени уходит на решение так называемой *проблемы выравнивания данных*. Возможно, у двух взаимосвязан-

ных временных рядов не точно совпадают индексы или у двух объектов DataFrame метки столбцов либо строк различны. Пользователи MATLAB, R и других языков, ориентированных на работу с матрицами, часто тратят немало сил, чтобы переформатировать данные, добиваясь идеальной выравненности. По моему опыту, выравнивать данные вручную (а еще хуже – проверять, что данные выровнены) слишком утомительно. А если этого не сделать, то возможны ошибки из-за комбинирования невыровненных данных.

В pandas принят другой подход – автоматическое выравнивание данных при выполнении арифметических операций. На практике это развязывает программисту руки и резко повышает его продуктивность. Для примера рассмотрим два объекта DataFrame, содержащих временные ряды котировок акций и объемов торгов:

```
In [16]: prices
Out[16]:
               AAPL    JNJ      SPX    XOM
2011-09-06  379.74  64.64  1165.24  71.15
2011-09-07  383.93  65.43  1198.62  73.65
2011-09-08  384.14  64.95  1185.90  72.82
2011-09-09  377.48  63.64  1154.23  71.01
2011-09-12  379.94  63.59  1162.27  71.84
2011-09-13  384.62  63.61  1172.87  71.65
2011-09-14  389.30  63.73  1188.68  72.64

In [17]: volume
Out[17]:
                AAPL       JNJ       XOM
2011-09-06  18173500  15848300  25416300
2011-09-07  12492000  10759700  23108400
2011-09-08  14839800  15551500  22434800
2011-09-09  20171900  17008200  27969100
2011-09-12  16697300  13448200  26205800
```

Пусть требуется вычислить средневзвешенную по объему цену с использованием всех имеющихся данных (приняв упрощающее предположение о том, что данные об объемах образуют подмножество данных о ценах). Поскольку pandas автоматически выравнивает данные при выполнении арифметических операций и игнорирует отсутствующие данные в функциях типа `sum`, мы можем записать решение этой задачи очень кратко:

```
In [18]: prices * volume
Out[18]:
                 AAPL         JNJ  SPX         XOM
2011-09-06  6901204890  1024434112  NaN  1808369745
2011-09-07  4796053560   704007171  NaN  1701933660
2011-09-08  5700560772  1010069925  NaN  1633702136
2011-09-09  7614488812  1082401848  NaN  1986085791
2011-09-12  6343972162   855171038  NaN  1882624672
2011-09-13         NaN         NaN  NaN         NaN
2011-09-14         NaN         NaN  NaN         NaN

In [19]: vwap = (prices * volume).sum() / volume.sum()

In [20]: vwap             In [21]: vwap.dropna()
```

```
Out[20]:                    Out[21]:
AAPL  380.655181            AAPL  380.655181
JNJ    64.394769            JNJ    64.394769
SPX          NaN            XOM    72.024288
XOM    72.024288
```

Поскольку символа `SPX` в ряду `volume` нет, можно явно отбросить его. Если вы хотите выравнивать данные вручную, то можете использовать метод объекта DataFrame `align`, который возвращает переиндексированные версии обоих объектов:

```
In [22]: prices.align(volume, join='inner')
Out[22]:
(             AAPL    JNJ    XOM
2011-09-06  379.74  64.64  71.15
2011-09-07  383.93  65.43  73.65
2011-09-08  384.14  64.95  72.82
2011-09-09  377.48  63.64  71.01
2011-09-12  379.94  63.59  71.84,
                AAPL       JNJ       XOM
2011-09-06  18173500  15848300  25416300
2011-09-07  12492000  10759700  23108400
2011-09-08  14839800  15551500  22434800
2011-09-09  20171900  17008200  27969100
2011-09-12  16697300  13448200  26205800)
```

Еще одна незаменимая вещь – конструирование DataFrame из коллекции объектов Series, возможно, проиндексированных по-разному:

```
In [23]: s1 = Series(range(3), index=['a', 'b', 'c'])

In [24]: s2 = Series(range(4), index=['d', 'b', 'c', 'e'])

In [25]: s3 = Series(range(3), index=['f', 'a', 'c'])

In [26]: DataFrame({'one': s1, 'two': s2, 'three': s3})
Out[26]:
   one  three  two
a    0      1  NaN
b    1    NaN    1
c    2      2    2
d  NaN    NaN    0
e  NaN    NaN    3
f  NaN      0  NaN
```

Конечно, можно явно задать индекс результата, отбросив лишние данные:

```
In [27]: DataFrame({'one': s1, 'two': s2, 'three': s3}, index=list('face'))
Out[27]:
    one  three  two
f   NaN      0  NaN
a     0      1  NaN
c     2      2    2
e   NaN    NaN    3
```

Операции над временными рядами с различной частотой

В экономике временные ряды часто имеют частоту год, квартал, месяц, день или какую-то нестандартную. Встречаются и ряды, лишенные какой-либо регулярности; например, текущие сведения о доходности акций могут поступать в любое время. Два основных инструмента преобразования частоты и выравнивания – методы `resample` м `reindex`. Метод `resample` преобразует данные к указанной фиксированной частоте, а `reindex` согласует данные с новым индексом. Оба поддерживают факультативную интерполяцию (например, прямое восполнение):

Рассмотрим небольшой временной ряд с недельной частотой:

```
In [28]: ts1 = Series(np.random.randn(3),
   ....:          index=pd.date_range('2012-6-13', periods=3, freq='W-WED'))

In [29]: ts1
Out[29]:
2012-06-13   -1.124801
2012-06-20    0.469004
2012-06-27   -0.117439
Freq: W-WED
```

Если передискретизировать его на частоту рабочих дней (с понедельника по пятницу), то появятся пропуски для дней, за которые нет данных:

```
In [30]: ts1.resample('B')
Out[30]:
2012-06-13  -1.124801
2012-06-14         NaN
2012-06-15         NaN
2012-06-18         NaN
2012-06-19         NaN
2012-06-20  0.469004
2012-06-21         NaN
2012-06-22         NaN
2012-06-25         NaN
2012-06-26         NaN
2012-06-27  -0.117439
Freq: B
```

Разумеется, задав параметр `fill_method='ffill'`, мы восполним эти пропуски. Это обычная практика, когда частота данными невелика, – вычислить временной ряд, в котором значения в каждой временной метке совпадают с последним предшествующим ей известным значением, т. е. берутся по принципу «по состоянию на»:

```
In [31]: ts1.resample('B', fill_method='ffill')
Out[31]:
2012-06-13  -1.124801
2012-06-14  -1.124801
2012-06-15  -1.124801
```

```
2012-06-18  -1.124801
2012-06-19  -1.124801
2012-06-20   0.469004
2012-06-21   0.469004
2012-06-22   0.469004
2012-06-25   0.469004
2012-06-26   0.469004
2012-06-27  -0.117439
Freq: B
```

На практике повышающая передискретизация от меньшей регулярной частоты к большей – приемлемое решение, но в более общем случае нерегулярных временных рядов аппроксимация может оказаться плохой. Рассмотрим нерегулярный временной ряд в том же промежутке времени:

```
In [32]: dates = pd.DatetimeIndex(['2012-6-12', '2012-6-17', '2012-6-18',
   ....:                           '2012-6-21', '2012-6-22', '2012-6-29'])

In [33]: ts2 = Series(np.random.randn(6), index=dates)

In [34]: ts2
Out[34]:
2012-06-12  -0.449429
2012-06-17   0.459648
2012-06-18  -0.172531
2012-06-21   0.835938
2012-06-22  -0.594779
2012-06-29   0.027197
```

Допустим, мы хотим добавить в `ts2` значения «по состоянию» в `ts1` (прямое восполнение). Один из вариантов – передискретизировать оба ряда на одну и ту же регулярную частоту, а затем произвести добавление, но если желательно сохранить индекс дат в `ts2`, то более точное решение даст метод `reindex`:

```
In [35]: ts1.reindex(ts2.index, method='ffill')
Out[35]:
2012-06-12        NaN
2012-06-17  -1.124801
2012-06-18  -1.124801
2012-06-21   0.469004
2012-06-22   0.469004
2012-06-29  -0.117439

In [36]: ts2 + ts1.reindex(ts2.index, method='ffill')
Out[36]:
2012-06-12       NaN
2012-06-17  -0.665153
2012-06-18  -1.297332
2012-06-21   1.304942
2012-06-22  -0.125775
2012-06-29  -0.090242
```

Использование периодов в качестве временных меток

Периоды (представляющие промежутки времени) дают другой способ работы с временными рядами разной частоты, особенно с финансовыми или экономическими рядами, содержащими годовые или квартальные данные и следующими определенным соглашениям об отчетности. Например, компания может публиковать данные о чистой прибыли за квартал в предположении, что финансовый год заканчивается в июне, т. е. с частотой `Q-JUN`. Рассмотрим два макроэкономических временных ряда, относящихся в ВВП и инфляции:

```
In [37]: gdp = Series([1.78, 1.94, 2.08, 2.01, 2.15, 2.31, 2.46],
    ....:             index=pd.period_range('1984Q2', periods=7, freq='Q-SEP'))

In [38]: infl = Series([0.025, 0.045, 0.037, 0.04],
    ....:              index=pd.period_range('1982', periods=4, freq='A-DEC'))

In [39]: gdp                  In [40]: infl
Out[39]:                      Out[40]:
1984Q2   1.78                 1982 0.025
1984Q3   1.94                 1983 0.045
1984Q4   2.08                 1984 0.037
1985Q1   2.01                 1985 0.040
1985Q2   2.15                 Freq: A-DEC
1985Q3   2.31
1985Q4   2.46
Freq: Q-SEP
```

В отличие от временных рядов с временными метками, операции между временными рядами, индексированными периодами разной частоты, невозможны без явных преобразований. В данном случае, если мы знаем, что значения `infl` замерялись в конце каждого года, то можем преобразовать этот ряд к частоте `Q-SEP`, чтобы периоды в обоих рядах совпали:

```
In [41]: infl_q = infl.asfreq('Q-SEP', how='end')

In [42]: infl_q
Out[42]:
1983Q1   0.025
1984Q1   0.045
1985Q1   0.037
1986Q1   0.040
Freq: Q-SEP
```

Этот временной ряд можно затем переиндексировать с прямым восполнением, чтобы привести его в соответствие с `gdp`:

```
In [43]: infl_q.reindex(gdp.index, method='ffill')
Out[43]:
1984Q2   0.045
1984Q3   0.045
1984Q4   0.045
1985Q1   0.037
```

```
1985Q2   0.037
1985Q3   0.037
1985Q4   0.037
Freq: Q-SEP
```

Время суток и выборка данных «по состоянию на»

Пусть имеется длинный временной ряд, содержащий внутридневные данные о рынках, и мы хотим извлечь цены в конкретное время суток за каждый день, представленный в ряде. Но что, если данные нерегулярны, и в интересующее нас время нет наблюдений? На практике такая задача может потребовать переформатирования данных, которое при недостаточной аккуратности может привести к ошибкам. Для иллюстрации рассмотрим такой пример:

```
# Создать внутридневной диапазон дат и временной ряд
In [44]: rng = pd.date_range('2012-06-01 09:30', '2012-06-01 15:59',
   ....:                     freq='T')

# Создать ряд, содержащий данные в диапазоне 9:30-15:59
# и охватывающий 5 дней
In [45]: rng = rng.append([rng + pd.offsets.BDay(i) for i in range(1, 4)])

In [46]: ts = Series(np.arange(len(rng), dtype=float), index=rng)

In [47]: ts
Out[47]:
2012-06-01 09:30:00     0
2012-06-01 09:31:00     1
2012-06-01 09:32:00     2
2012-06-01 09:33:00     3
...
2012-06-06 15:56:00  1556
2012-06-06 15:57:00  1557
2012-06-06 15:58:00  1558
2012-06-06 15:59:00  1559
Length: 1560
```

Доступ по индексу с помощью объекта Python `datetime.time` возвращает данные в указанный момент времени:

```
In [48]: from datetime import time

In [49]: ts[time(10, 0)]
Out[49]:
2012-06-01 10:00:00    30
2012-06-04 10:00:00   420
2012-06-05 10:00:00   810
2012-06-06 10:00:00  1200
```

Под капотом используется метод экземпляра `at_time` (который имеется у объектов временных рядов и DataFrame):

```
In [50]: ts.at_time(time(10, 0))
Out[50]:
```

```
2012-06-01 10:00:00    30
2012-06-04 10:00:00   420
2012-06-05 10:00:00   810
2012-06-06 10:00:00  1200
```

Можно также выбрать значения в промежутке между двумя моментами времени, воспользовавшись методом `between_time`:

```
In [51]: ts.between_time(time(10, 0), time(10, 1))
Out[51]:
2012-06-01 10:00:00    30
2012-06-01 10:01:00    31
2012-06-04 10:00:00   420
2012-06-04 10:01:00   421
2012-06-05 10:00:00   810
2012-06-05 10:01:00   811
2012-06-06 10:00:00  1200
2012-06-06 10:01:00  1201
```

Как уже отмечалось, бывает, что в некоторый момент времени, например в 10 утра, никаких данных нет, но хотелось бы получить последнее известное значение, предшествующее 10 утра:

```
# Случайным образом записать NA в большинство элементов временного ряда
In [53]: indexer = np.sort(np.random.permutation(len(ts))[700:])

In [54]: irr_ts = ts.copy()

In [55]: irr_ts[indexer] = np.nan

In [56]: irr_ts['2012-06-01 09:50':'2012-06-01 10:00']
Out[56]:
2012-06-01 09:50:00   20
2012-06-01 09:51:00  NaN
2012-06-01 09:52:00   22
2012-06-01 09:53:00   23
2012-06-01 09:54:00  NaN
2012-06-01 09:55:00   25
2012-06-01 09:56:00  NaN
2012-06-01 09:57:00  NaN
2012-06-01 09:58:00  NaN
2012-06-01 09:59:00  NaN
2012-06-01 10:00:00  NaN
```

Передав массив временных меток методу `asof`, мы получим массив последних известных (отличных от NA) значений в момент, совпадающий или предшествующий каждой метке. Построим диапазон дат, соответствующих 10 утра каждого дня, и передадим его методу `asof`:

```
In [57]: selection = pd.date_range('2012-06-01 10:00', periods=4, freq='B')

In [58]: irr_ts.asof(selection)
Out[58]:
```

```
2012-06-01 10:00:00    25
2012-06-04 10:00:00   420
2012-06-05 10:00:00   810
2012-06-06 10:00:00  1197
Freq: B
```

Сращивание источников данных

В главе 7 я описал ряд стратегий слияния двух наборов данных. В финансовом или экономическом контексте существует несколько широко распространенных случаев:

- переключение с одного источника данных (временного ряда или коллекции временных рядов) на другой в некоторый момент времени;
- вставка в начало, середину или конец одного временного ряда отсутствующих значений, заимствованных из другого ряда;
- полная замена данных для некоторого подмножества символов (стран, торговых кодов акций и т. д.).

В первом случае необходимо срастить два объекта TimeSeries или DataFrame методом `pandas.concat`:

```
In [59]: data1 = DataFrame(np.ones((6, 3), dtype=float),
   ....:                   columns=['a', 'b', 'c'],
   ....:                   index=pd.date_range('6/12/2012', periods=6))

In [60]: data2 = DataFrame(np.ones((6, 3), dtype=float) * 2,
   ....:                   columns=['a', 'b', 'c'],
   ....:                   index=pd.date_range('6/13/2012', periods=6))

In [61]: spliced = pd.concat([data1.ix[:'2012-06-14'],
   ....:                      data2.ix['2012-06-15':]])

In [62]: spliced
Out[62]:
            a  b  c
2012-06-12  1  1  1
2012-06-13  1  1  1
2012-06-14  1  1  1
2012-06-15  2  2  2
2012-06-16  2  2  2
2012-06-17  2  2  2
2012-06-18  2  2  2
```

Рассмотрим другой пример, когда в `data1` отсутствует временной ряд, имеющийся в `data2`:

```
In [63]: data2 = DataFrame(np.ones((6, 4), dtype=float) * 2,
   ....:                   columns=['a', 'b', 'c', 'd'],
   ....:                   index=pd.date_range('6/13/2012', periods=6))

In [64]: spliced = pd.concat([data1.ix[:'2012-06-14'],
   ....:                      data2.ix['2012-06-15':]])

In [65]: spliced
```

```
Out[65]:
            a  b  c    d
2012-06-12  1  1  1  NaN
2012-06-13  1  1  1  NaN
2012-06-14  1  1  1  NaN
2012-06-15  2  2  2    2
2012-06-16  2  2  2    2
2012-06-17  2  2  2    2
2012-06-18  2  2  2    2
```

С помощью метода `combine_first` мы можем вставить данные перед точкой сращения и тем самым расширить историю в столбце `'d'`:

```
In [66]: spliced_filled = spliced.combine_first(data2)

In [67]: spliced_filled
Out[67]:
            a  b  c    d
2012-06-12  1  1  1  NaN
2012-06-13  1  1  1    2
2012-06-14  1  1  1    2
2012-06-15  2  2  2    2
2012-06-16  2  2  2    2
2012-06-17  2  2  2    2
2012-06-18  2  2  2    2
```

Поскольку `data2` не содержит значения на дату `2012-06-12`, то этот день остается незаполненным. В объекте DataFrame имеется родственный метод `update` для выполнения обновления на месте. Если вы хотите, чтобы он только заполнял пропуски, задайте параметр `overwrite=False`:

```
In [68]: spliced.update(data2, overwrite=False)
In [69]: spliced
Out[69]:
            a  b  c    d
2012-06-12  1  1  1  NaN
2012-06-13  1  1  1    2
2012-06-14  1  1  1    2
2012-06-15  2  2  2    2
2012-06-16  2  2  2    2
2012-06-17  2  2  2    2
2012-06-18  2  2  2    2
```

Чтобы заменить данные для подмножества символов, можно воспользоваться любым из описанных выше приемов, но иногда проще напрямую задать столбцы с помощью доступа к DataFrame по индексу:

```
In [70]: cp_spliced = spliced.copy()

In [71]: cp_spliced[['a', 'c']] = data1[['a', 'c']]

In [72]: cp_spliced
Out[72]:
```

```
              a    b    c    d
2012-06-12    1    1    1  NaN
2012-06-13    1    1    1    2
2012-06-14    1    1    1    2
2012-06-15    1    2    1    2
2012-06-16    1    2    1    2
2012-06-17    1    2    1    2
2012-06-18  NaN    2  NaN    2
```

Индексы доходности и кумулятивная доходность

В контексте финансов под *доходностью* обычно понимают процентное изменение стоимости актива. Рассмотрим данные о котировках акций Apple в 2011 и 2012 году:

```
In [73]: import pandas.io.data as web

In [74]: price = web.get_data_yahoo('AAPL', '2011-01-01')['Adj Close']

In [75]: price[-5:]
Out[75]:
Date
2012-07-23  603.83
2012-07-24  600.92
2012-07-25  574.97
2012-07-26  574.88
2012-07-27  585.16
Name: Adj Close
```

В случае компании Apple, которая не платит дивиденды, для вычисления кумулятивной процентной доходности за период между двумя моментами времени необходимо вычислить процентное изменение цены:

```
In [76]: price['2011-10-03'] / price['2011-3-01'] - 1
Out[76]: 0.072399874037388123
```

Для других акций, по которым дивиденды выплачиваются, вычисление дохода от владения одной акцией несколько сложнее. Однако в использованных здесь скорректированных значениях цены закрытия учтены дробления и дивиденды. В любом случае принято вычислять *индекс доходности* – временной ряд, показывающий ценность единичного инвестирования (скажем, на сумму в один доллар). В основу вычисления индекса доходности можно положить различные предположения; например, одни предпочитают реинвестировать прибыль, другие – нет. В случае Apple мы можем вычислить индекс доходности с помощью метода `cumprod`:

```
In [77]: returns = price.pct_change()

In [78]: ret_index = (1 + returns).cumprod()

In [79]: ret_index[0] = 1 # Set first value to 1

In [80]: ret_index
```

```
Out[80]:
Date
2011-01-03  1.000000
2011-01-04  1.005219
2011-01-05  1.013442
2011-01-06  1.012623
...
2012-07-24  1.823346
2012-07-25  1.744607
2012-07-26  1.744334
2012-07-27  1.775526
Length: 396
```

Имея индекс доходности, уже легко вычислить кумулятивную доходность при заданной периодичности начисления:

```
In [81]: m_returns = ret_index.resample('BM', how='last').pct_change()
In [82]: m_returns['2012']
Out[82]:
Date
2012-01-31   0.127111
2012-02-29   0.188311
2012-03-30   0.105284
2012-04-30  -0.025969
2012-05-31  -0.010702
2012-06-29   0.010853
2012-07-31   0.001986
Freq: BM
```

Разумеется, в этом простом случае (отсутствие дивидендов и прочих корректировок) доходность можно было бы вычислить и по суточным процентным изменениям, произведя передискретизацию с агрегированием (в данном случае с преобразованием в периоды):

```
In [83]: m_rets = (1 + returns).resample('M', how='prod', kind='period') - 1

In [84]: m_rets['2012']
Out[84]:
Date
2012-01   0.127111
2012-02   0.188311
2012-03   0.105284
2012-04  -0.025969
2012-05  -0.010702
2012-06   0.010853
2012-07   0.001986
Freq: M
```

Если бы мы знали даты выплаты дивидендов и процентные ставки, то для включения их в общую суточную доходность нужно было бы написать:

```
returns[dividend_dates] += dividend_pcts
```

Групповые преобразования и анализ

В главе 9 мы познакомились с основами вычисления групповых статистик и применением собственных преобразований к группам в наборе данных.

Рассмотрим коллекцию гипотетических портфелей ценных бумаг. Сначала я случайным образом сгенерирую *генеральную совокупность* из 2000 торговых кодов:

```
import random; random.seed(0)
import string

N = 1000
def rands(n):
  choices = string.ascii_uppercase
  return ''.join([random.choice(choices) for _ in xrange(n)])
tickers = np.array([rands(5) for _ in xrange(N)])
```

Затем я создам объект DataFrame с 3 столбцами, представляющими три гипотетических случайных портфели, содержащих подмножества этих торговых кодов:

```
M = 500
df = DataFrame({'Momentum' : np.random.randn(M) / 200 + 0.03,
                'Value' : np.random.randn(M) / 200 + 0.08,
                'ShortInterest' : np.random.randn(M) / 200 - 0.02},
                index=tickers[:M])
```

Далее я создам случайную классификацию кодов по отраслям промышленности. Чтобы не усложнять пример, я ограничусь двумя отраслями и сохраню соответствие в объекте Series:

```
ind_names = np.array(['FINANCIAL', 'TECH'])
sampler = np.random.randint(0, len(ind_names), N)
industries = Series(ind_names[sampler], index=tickers,
                    name='industry')
```

Теперь мы можем сгруппировать по `industries` и выполнить агрегирование по группам и преобразования:

```
In [90]: by_industry = df.groupby(industries)

In [91]: by_industry.mean()
Out[91]:
           Momentum  ShortInterest     Value
industry
FINANCIAL  0.029485      -0.020739  0.079929
TECH       0.030407      -0.019609  0.080113

In [92]: by_industry.describe()
Out[92]:
                   Momentum  ShortInterest       Value
industry
FINANCIAL count  246.000000     246.000000  246.000000
```

```
            mean     0.029485      -0.020739    0.079929
            std      0.004802       0.004986    0.004548
            min      0.017210      -0.036997    0.067025
            25%      0.026263      -0.024138    0.076638
            50%      0.029261      -0.020833    0.079804
            75%      0.032806      -0.017345    0.082718
            max      0.045884      -0.006322    0.093334
TECH        count  254.000000     254.000000  254.000000
            mean     0.030407      -0.019609    0.080113
            std      0.005303       0.005074    0.004886
            min      0.016778      -0.032682    0.065253
            25%      0.026456      -0.022779    0.076737
            50%      0.030650      -0.019829    0.080296
            75%      0.033602      -0.016923    0.083353
            max      0.049638      -0.003698    0.093081
```

Определив функции преобразования, легко преобразовать эти портфели по отраслям промышленности. Например, при формировании портфеля ценных бумаг части применяется внутриотраслевая стандартизация:

```
# Внутриотраслевая стандартизация
def zscore(group):
  return (group - group.mean()) / group.std()

df_stand = by_industry.apply(zscore)
```

Легко убедиться, что при таком преобразовании среднее для каждой отрасли равно 0, а стандартное отклонение – 1:

```
In [94]: df_stand.groupby(industries).agg(['mean', 'std'])
Out[94]:
             Momentum       ShortInterest        Value
                 mean std            mean std     mean std
industry
FINANCIAL           0   1               0   1        0   1
TECH               -0   1              -0   1       -0   1
```

Встроенные преобразования, например `rank`, записываются более кратко:

```
# Внутриотраслевое ранжирование в порядке убывания
In [95]: ind_rank = by_industry.rank(ascending=False)

In [96]: ind_rank.groupby(industries).agg(['min', 'max'])
Out[96]:
             Momentum       ShortInterest        Value
                  min max             min max      min max
industry
FINANCIAL           1 246               1 246        1 246
TECH                1 254               1 254        1 254
```

В финансовой математике «ранжирование и стандартизация» – часто встречающаяся последовательность преобразований. Для этого можно сцепить вместе функции `rank` и `zscore`:

```
# Ранжирование и стандартизация по отраслям
In [97]: by_industry.apply(lambda x: zscore(x.rank()))
Out[97]:
<class 'pandas.core.frame.DataFrame'>
Index: 500 entries, VTKGN to PTDQE
Data columns:
Momentum        500  non-null values
ShortInterest   500  non-null values
Value           500  non-null values
dtypes: float64(3)
```

Оценка воздействия групповых факторов

Факторный анализ – один из методов алгоритмического управления портфелем. Портфельные активы и показатели (прибыли и убытки) анализируются с применением одного или нескольких *факторов* (например, факторов риска), представленных в виде портфеля весов. Например, параллельная динамика стоимости ценной бумаги и основного индекса (например, S&P 500) характеризуется *бета-коэффициентом*, это один из типичных факторов риска. Рассмотрим пример гипотетического портфеля, построенного из трех случайных факторов (их обычно называют *факторными нагрузками*) и некоторых весов:

```
from numpy.random import rand
fac1, fac2, fac3 = np.random.rand(3, 1000)

ticker_subset = tickers.take(np.random.permutation(N)[:1000])

# Взвешенная сумма факторов плюс шум
port = Series(0.7 * fac1 - 1.2 * fac2 + 0.3 * fac3 + rand(1000),
              index=ticker_subset)
factors = DataFrame({'f1': fac1, 'f2': fac2, 'f3': fac3},
                    index=ticker_subset)
```

Векторные корреляции между каждым фактором и портфелем мало о чем говорят:

```
In [99]: factors.corrwith(port)
Out[99]:
f1   0.402377
f2  -0.680980
f3   0.168083
```

Стандартный способ оценки воздействия факторов – регрессия методом наименьших квадратов; передав функции `pandas.ols` объект `factors` в качестве объясняющей переменной, мы сможем оценить воздействие по всему набору торговых кодов:

```
In [100]: pd.ols(y=port, x=factors).beta
Out[100]:
f1   0.761789
f2  -1.208760
f3   0.289865
intercept 0.484477
```

Как видим, исходные веса факторов почти удалось восстановить, потому что величина добавленного в портфель шума была не слишком велика. С помощью метода `groupby` можно оценить воздействие для каждой отрасли. Для этого напишем такую функцию:

```
def beta_exposure(chunk, factors=None):
  return pd.ols(y=chunk, x=factors).beta
```

Затем сгруппируем по `industries` и применим эту функцию, передав объект DataFrame, содержащий факторные нагрузки:

```
In [102]: by_ind = port.groupby(industries)

In [103]: exposures = by_ind.apply(beta_exposure, factors=factors)

In [104]: exposures.unstack()
Out[104]:
                  f1        f2        f3  intercept
industry
FINANCIAL  0.790329  -1.182970  0.275624   0.455569
TECH       0.740857  -1.232882  0.303811   0.508188
```

Децильный и квартильный анализ

Анализ данных на основе выборочных квантилей – еще один важный инструмент финансового аналитика. Например, исполнение портфеля акций можно разложить на квартили (четыре части одинакового размера), исходя из отношения рыночной цены каждой акции к чистой прибыли. Благодаря сочетанию функций `pandas.qcut` и `groupby` квантильный анализ выполняется довольно просто. В качестве примера рассмотрим простую торговую стратегию следования за трендом торгуемого инвестиционного фонда SPY, отраженного в индексе S&P 500. Скачать историю изменения цен можно с сайта Yahoo! Finance:

```
In [105]: import pandas.io.data as web

In [106]: data = web.get_data_yahoo('SPY', '2006-01-01')

In [107]: data
Out[107]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1655 entries, 2006-01-03 00:00:00 to 2012-07-27 00:00:00
Data columns:
Open       1655  non-null values
High       1655  non-null values
Low        1655  non-null values
Close      1655  non-null values
Volume     1655  non-null values
Adj Close  1655  non-null values
dtypes: float64(5), int64(1)
```

Теперь вычислим суточную доходность и напишем функцию для преобразования доходности в трендовый индикатор, построенный по скользящей сумме с отставанием:

```
px = data['Adj Close']
returns = px.pct_change()

def to_index(rets):
  index = (1 + rets).cumprod()
  first_loc = max(index.notnull().argmax() - 1, 0)
  index.values[first_loc] = 1
  return index

def trend_signal(rets, lookback, lag):
  signal = pd.rolling_sum(rets, lookback, min_periods=lookback - 5)
  return signal.shift(lag)
```

С помощью этой функции можем создать и протестировать (наивную) стратегию торговли по этому трендовому индикатору в каждую пятницу:

```
In [109]: signal = trend_signal(returns, 100, 3)

In [110]: trade_friday = signal.resample('W-FRI')
    ....:                       .resample('B', fill_method='ffill')

In [111]: trade_rets = trade_friday.shift(1) * returns
```

Затем можно преобразовать доходность этой стратегии в коэффициент доходности и нанести его на график (рис. 11.1).

```
In [112]: to_index(trade_rets).plot()
```

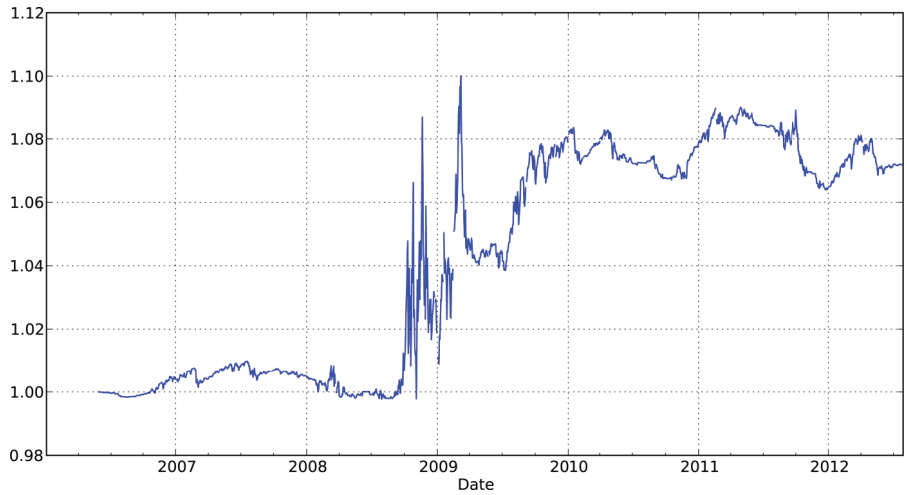


Рис. 11.1. Индекс доходности стратегии динамичной торговли по тренду фонда SPY

Пусть требуется разложить исполнение стратегии на более и менее волатильные периоды торговли. Простым показателем волатильности является стандартное отклонение, посчитанное за последний год, и мы можем вычислить коэффициенты Шарпа для оценки доходности к риску в различных режимах волатильности:

```
vol = pd.rolling_std(returns, 250, min_periods=200) * np.sqrt(250)

def sharpe(rets, ann=250):
  return rets.mean() / rets.std() * np.sqrt(ann)
```

Затем, разбив `vol` на квартили с помощью `qcut` и агрегировав с помощью `sharpe`, получаем:

```
In [114]: trade_rets.groupby(pd.qcut(vol, 4)).agg(sharpe)
Out[114]:
[0.0955, 0.16]   0.490051
(0.16, 0.188]    0.482788
(0.188, 0.231]  -0.731199
(0.231, 0.457]   0.570500
```

Отсюда видно, что стратегия дала наилучшие результаты в те периоды, когда волатильность была наиболее высокой.

Другие примеры приложений

Ниже приводится несколько дополнительных примеров.

Стохастический граничный анализ

В этом разделе я опишу упрощенный перекрестный динамичный портфель и покажу, как можно исследовать сетку параметров модели. Для начала загружу историю цен для портфеля акций финансовых и технологических компаний:

```
names = ['AAPL', 'GOOG', 'MSFT', 'DELL', 'GS', 'MS', 'BAC', 'C']
def get_px(stock, start, end):
  return web.get_data_yahoo(stock, start, end)['Adj Close']
px = DataFrame({n: get_px(n, '1/1/2009', '6/1/2012') for n in names})
```

Нетрудно построить график кумулятивной доходности каждой акции (рис. 11.2).

```
In [117]: px = px.asfreq('B').fillna(method='pad')

In [118]: rets = px.pct_change()

In [119]: ((1 + rets).cumprod() - 1).plot()
```

Для построения портфеля вычислим индикатор темпа за некоторый прошлый период, а затем выполним ранжирование в порядке убывания и стандартизацию:

```
def calc_mom(price, lookback, lag):
  mom_ret = price.shift(lag).pct_change(lookback)
  ranks = mom_ret.rank(axis=1, ascending=False)
  demeaned = ranks - ranks.mean(axis=1)
  return demeaned / demeaned.std(axis=1)
```

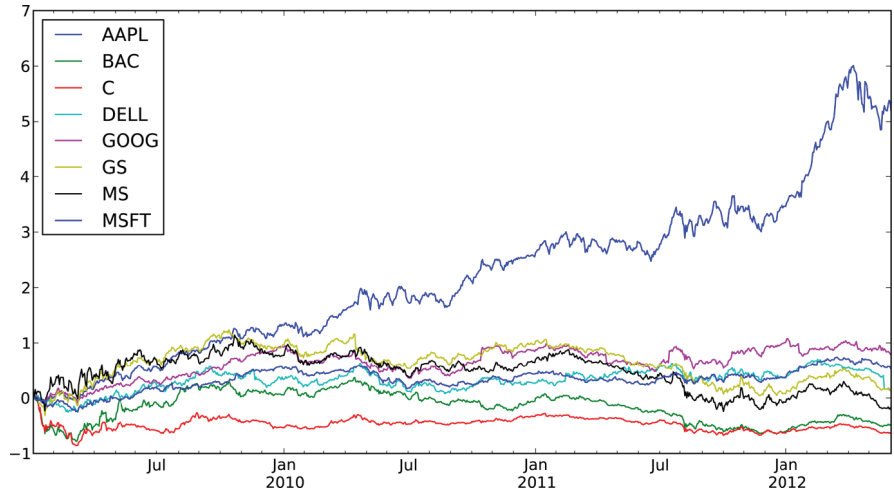


Рис. 11.2. Кумулятивная доходность для каждой акции

Имея эту функцию преобразования, мы можем написать функцию ретроспективного тестирования стратегия, которая вычисляет портфель за некоторый прошлый период и период владения (количество дней между торгами) и возвращает совокупный коэффициент Шарпа:

```
compound = lambda x : (1 + x).prod() - 1
daily_sr = lambda x: x.mean() / x.std()

def strat_sr(prices, lb, hold):
  # Вычислить веса портфеля
  freq = '%dB' % hold
  port = calc_mom(prices, lb, lag=1)
  daily_rets = prices.pct_change()

  # Вычислить доходность портфеля
  port = port.shift(1).resample(freq, how='first')
  returns = daily_rets.resample(freq, how=compound)
  port_rets = (port * returns).sum(axis=1)

  return daily_sr(port_rets) * np.sqrt(252 / hold)
```

Если передать этой функции цены и комбинацию параметров, то она вернет скалярное значение:

```
In [122]: strat_sr(px, 70, 30)
Out[122]: 0.27421582756800583
```

Далее можно вычислить значения функции `strat_sr` на сетке параметров модели, сохранить их в словаре `defaultdict` и, наконец, поместить результаты в DataFrame:

```
from collections import defaultdict
lookbacks = range(20, 90, 5)
holdings = range(20, 90, 5)
dd = defaultdict(dict)
for lb in lookbacks:
  for hold in holdings:
     dd[lb][hold] = strat_sr(px, lb, hold)

ddf = DataFrame(dd)
ddf.index.name = 'Holding Period'
ddf.columns.name = 'Lookback Period'
```

Чтобы наглядно представить результаты и понять, что происходит, я написал функцию, которая с помощью matplotlib строит тепловую карту с пояснениями:

```
import matplotlib.pyplot as plt

def heatmap(df, cmap=plt.cm.gray_r):
  fig = plt.figure()
  ax = fig.add_subplot(111)
  axim = ax.imshow(df.values, cmap=cmap, interpolation='nearest')
  ax.set_xlabel(df.columns.name)
  ax.set_xticks(np.arange(len(df.columns)))
  ax.set_xticklabels(list(df.columns))
  ax.set_ylabel(df.index.name)
  ax.set_yticks(np.arange(len(df.index)))
  ax.set_yticklabels(list(df.index))
  plt.colorbar(axim)
```

Получив результаты ретроспективного анализа, эта функция построила график, показанный на рис. 11.3:

```
In [125]: heatmap(ddf)
```

Роллинг фьючерсных контрактов

Слово «фьючерс» постоянно встречается, когда говорят о производных контрактах; это договор о поставке определенного актива (нефти, золота или акций из индекса Британской фондовой биржи FTSE 100) в определенный день. На практике моделирование и торговля фьючерсными контрактами на акции, валюты, сырьевые товары, облигации и прочие категории активов осложняется ограниченностью контракта во времени. Например, в любой момент времени для некоторого типа фьючерсов (скажем, на серебро или медь) может торговаться несколько контрактов с различными датами *окончания срока действия*. Во многих случаях наиболее ликвидным (с наибольшим объемом торгов и наименьшей разницей между ценами продавца и покупателя) будет фьючерсный контракт с ближайшей датой окончания (*близкий* контракт).

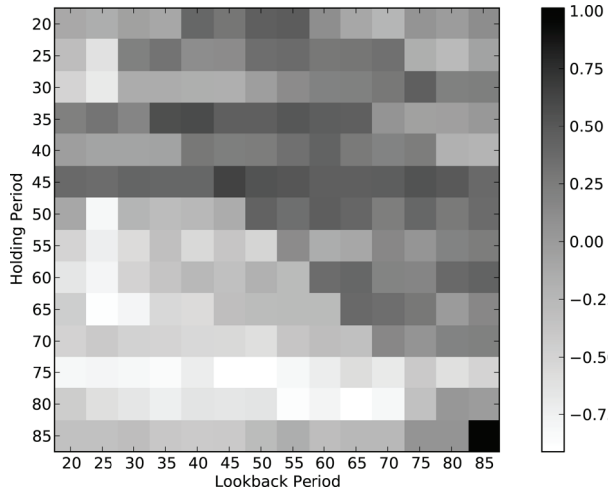


Рис. 11.3. Тепловая карта коэффициента Шарпа для динамичной стратегии торговли по тренду (чем коэффициент выше, тем лучше) при различных прошлых периодах и периодах владения

Для моделирования и прогнозирования было бы гораздо проще работать с *непрерывным* индексом доходности, который показывает прибыли и убытки при постоянном владении ближайшим контрактом. Переход от контракта с истекающим сроком действия к следующему (или *отдаленному*) контракту называется роллингом. Вычисление непрерывного ряда фьючерсов по данным отдельного контракта – отнюдь не простая задача, для решения которой обычно требуется глубокое понимание рынка и принципов торговли производными инструментами. Вот пример практического вопроса: когда и как быстро следует переходить от контракта с истекающим сроком действия к следующему? Ниже описывается один такой процесс.

Я буду работать с масштабированными ценами торгуемого инвестиционного фонда SPY как приближением к индексу S&P 500:

```
In [127]: import pandas.io.data as web

# Аппроксимировать цену из индекса S&P 500
In [128]: px = web.get_data_yahoo('SPY')['Adj Close'] * 10

In [129]: px
Out[129]:
Date
2011-08-01  1261.0
2011-08-02  1228.8
2011-08-03  1235.5
...
2012-07-25  1339.6
2012-07-26  1361.7
2012-07-27  1386.8
Name: Adj Close, Length: 251
```

Теперь несколько подготовительных действий. Вставлю в объект Series два фьючерсных контракта на акции компаний из S&P 500 вместе с датами окончания срока действия:

```
from datetime import datetime
expiry = {'ESU2': datetime(2012, 9, 21),
          'ESZ2': datetime(2012, 12, 21)}
expiry = Series(expiry).order()
```

Теперь `expiry` выглядит следующим образом:

```
In [131]: expiry
Out[131]:
ESU2   2012-09-21 00:00:00
ESZ2   2012-12-21 00:00:00
```

Далее я воспользовался ценами с сайта Yahoo! Finance, а также случайным блужданием и добавлением шума для моделирования поведения обоих контрактов в будущем:

```
np.random.seed(12347)
N = 200
walk = (np.random.randint(0, 200, size=N) - 100) * 0.25
perturb = (np.random.randint(0, 20, size=N) - 10) * 0.25
walk = walk.cumsum()

rng = pd.date_range(px.index[0], periods=len(px) + N, freq='B')
near = np.concatenate([px.values, px.values[-1] + walk])
far = np.concatenate([px.values, px.values[-1] + walk + perturb])
prices = DataFrame({'ESU2': near, 'ESZ2': far}, index=rng)
```

В объекте `prices` сейчас хранятся два временных ряда для обоих контрактов, которые отличаются друг от друга на случайную величину:

```
In [133]: prices.tail()
Out[133]:
                ESU2      ESZ2
2013-04-16   1416.05   1417.80
2013-04-17   1402.30   1404.55
2013-04-18   1410.30   1412.05
2013-04-19   1426.80   1426.05
2013-04-22   1406.80   1404.55
```

Один из способов срастить оба временных ряда в один непрерывный ряд – построить весовую матрицу. У активных контрактов должен быть вес 1 до тех пор, пока не приблизится дата окончания срока действия. И теперь нужно принять какое-то решение о роллинге. Ниже приведена функция, которая вычисляет весовую матрицу с линейным затуханием на протяжении нескольких периодов на подходе к конечной дате:

```
def get_roll_weights(start, expiry, items, roll_periods=5):
  # start : начальная дата для вычисления весового объекта DataFrame
```

```
    # expiry : объект Series, сопоставляющий торговому коду дату
    #          окончания срока действия
    # items : последовательность имен контрактов
    dates = pd.date_range(start, expiry[-1], freq='B')
    weights = DataFrame(np.zeros((len(dates), len(items))),
    index=dates, columns=items)
    prev_date = weights.index[0]
    for i, (item, ex_date) in enumerate(expiry.iteritems()):
       if i < len(expiry) - 1:
          weights.ix[prev_date:ex_date - pd.offsets.BDay(), item] = 1
          roll_rng = pd.date_range(end=ex_date - pd.offsets.BDay(),
                                   periods=roll_periods + 1, freq='B')

          decay_weights = np.linspace(0, 1, roll_periods + 1)
          weights.ix[roll_rng, item] = 1 - decay_weights
          weights.ix[roll_rng, expiry.index[i + 1]] = decay_weights
       else:
          weights.ix[prev_date:, item] = 1

    prev_date = ex_date
    return weights
```

При приближении к дате окончания срока действия `ESU2` веса выглядят так:

```
In [135]: weights = get_roll_weights('6/1/2012', expiry, prices.columns)

In [136]: weights.ix['2012-09-12':'2012-09-21']
Out[136]:
             ESU2  ESZ2
2012-09-12   1.0   0.0
2012-09-13   1.0   0.0
2012-09-14   0.8   0.2
2012-09-17   0.6   0.4
2012-09-18   0.4   0.6
2012-09-19   0.2   0.8
2012-09-20   0.0   1.0
2012-09-21   0.0   1.0
```

Наконец, доходность продленного фьючерса равна взвешенной сумме доходностей обоих контрактов:

```
In [137]: rolled_returns = (prices.pct_change() * weights).sum(1)
```

Скользящая корреляция и линейная регрессия

Динамические модели играют важную роль в финансовом моделировании, поскольку их можно использовать для имитации торговых решений на протяжении исторического периода. Скользящие оконные и экспоненциально взвешенные функции временных рядов – примеры инструментов, применяемых в динамических моделях. Корреляция – это один из методов изучения параллельной динамики изменений временных рядов, описывающих торговлю активами. Функция `rolling_corr` из библиотеки pandas вызывается с двумя временными рядами на

входе и вычисляет скользящую оконную корреляцию. Для начала загружаю временные ряды с данными о ценах с сайта Yahoo! Finance и вычисляю суточную доходность:

```
aapl = web.get_data_yahoo('AAPL', '2000-01-01')['Adj Close']
msft = web.get_data_yahoo('MSFT', '2000-01-01')['Adj Close']

aapl_rets = aapl.pct_change()
msft_rets = msft.pct_change()
```

Затем вычисляю и строю график скользящей корреляции за один год (рис. 11.4):

```
In [140]: pd.rolling_corr(aapl_rets, msft_rets, 250).plot()
```

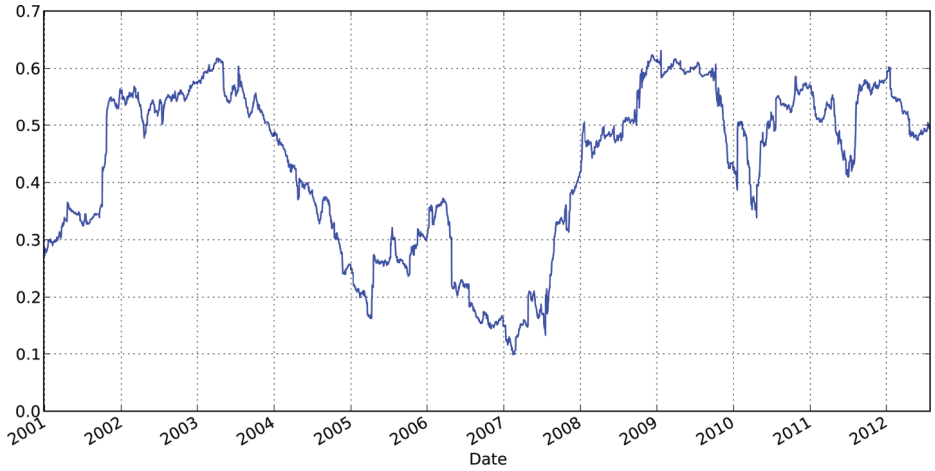


Рис. 11.4. Корреляция между Apple и Microsoft на протяжении одного года

Одна из проблем заключается в том, что корреляция между двумя активами не улавливает различий в волатильности. Регрессия методом наименьших квадратов дает другой способ моделирования динамической взаимосвязи между некоторой величиной и одним или несколькими прогностическими параметрами.

```
In [142]: model = pd.ols(y=aapl_rets, x={'MSFT': msft_rets}, window=250)

In [143]: model.beta
Out[143]:
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2913 entries, 2000-12-28 00:00:00 to 2012-07-27 00:00:00
Data columns:
MSFT       2913  non-null values
intercept  2913  non-null values
dtypes: float64(2)
In [144]: model.beta['MSFT'].plot()
```

Рис. 11.5. Бета-коэффициент (коэффициент регрессии обычным методом наименьших квадратов) между Apple и Microsoft за один год

Входящая в pandas функция `ols` реализует статическую и динамическую (по расширяющемуся или скользящему окну) регрессию обычным методом наименьших квадратов. Более развитые статистические и эконометрические модели можно найти в проекте statsmodels (*http://statsmodels.sourceforge.net*).

ГЛАВА 12. Дополнительные сведения о библиотеке NumPy

Внутреннее устройство объекта ndarray

Объект ndarray из библиотеки NumPy позволяет интерпретировать блок однородных данных (непрерывный или с шагом, подробнее об этом ниже) как многомерный массив. Мы уже видели, что тип данных, или `dtype`, определяет, как именно интерпретируются данные: как числа с плавающей точкой, целые, булевы или еще как-то.

Своей эффективностью ndarray отчасти обязан тому, что любой объект массива является *шаговым* (strided) представлением блока данных. Может возникнуть вопрос, как удается построить представление массива `arr[::2, ::-1]` без копирования данных. Дело в том, что объект ndarray – не просто блок памяти, дополненный знанием о типе в виде dtype; в нем еще хранится информация, позволяющая перемещаться по массиву шагами разного размера. Точнее, в реализации ndarray имеется:

- *указатель на данные*, т. е. на блок полученной от системы памяти;
- *тип данных*, или dtype;
- кортеж, описывающий *форму* массива. Например, у массива 10 × 5 будет форма `(10, 5)`:

```
In [8]: np.ones((10, 5)).shape
Out[8]: (10, 5)
```

- кортеж *шагов*, т. е. целых чисел, показывающих, на сколько байтов нужно сместиться, чтобы перейти к следующему элементу по некоторому измерению. Например, для типичного массива (организованного в соответствии с принятым в языке C соглашением, об этом ниже) 3 × 4 × 5 чисел типа `float64` (8-байтовых) кортеж шагов имеет вид `(160, 40, 8)`:

```
In [9]: np.ones((3, 4, 5), dtype=np.float64).strides
Out[9]: (160, 40, 8)
```

Хотя типичному пользователю NumPy редко приходится интересоваться шагами массива, они играют важнейшую роль в построении представлений массива

без копирования. Шаги могут быть даже отрицательными, что позволяет проходить массив в *обратном направлении*, как в случае среза вида `obj[::-1]` или `obj[:, ::-1]`.

На рис. 12.1 схематически показано внутреннее устройство ndarray.

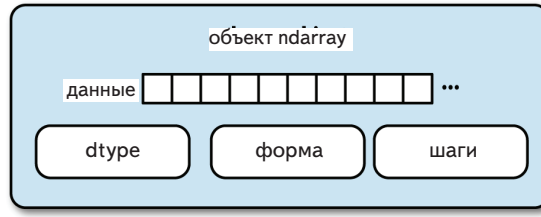


Рис. 12.1. Объект ndarray из библиотеки NumPy

Иерархия типов данных в NumPy

Иногда в программе необходимо проверить, что хранится в массиве: целые числа, числа с плавающей точкой, строки или объекты Python. Поскольку существует много типов с плавающей точкой (от `float16` до `float128`), для проверки того, что dtype присутствует в списке типов, приходится писать длинный код. По счастью, определены такие суперклассы, как `np.integer` или `np.floating`, которые можно использовать в сочетании с функцией `np.issubdtype`:

```
In [10]: ints = np.ones(10, dtype=np.uint16)

In [11]: floats = np.ones(10, dtype=np.float32)

In [12]: np.issubdtype(ints.dtype, np.integer)
Out[12]: True

In [13]: np.issubdtype(floats.dtype, np.floating)
Out[13]: True
```

Вывести все родительские классы данного типа dtype позволяет метод `mro`:

```
In [14]: np.float64.mro()
Out[14]:
[numpy.float64,
 numpy.floating,
 numpy.inexact,
 numpy.number,
 numpy.generic,
 float,
 object]
```

Большинству пользователей NumPy об этом знать необязательно, но иногда оказывается удобно. На рис. 12.2 показан граф наследования dtype[1].

[1] В именах некоторых типов dtype присутствуют знаки подчеркивания. Они нужны, чтобы избежать конфликтов между именами типов NumPy и встроенных типов Python.

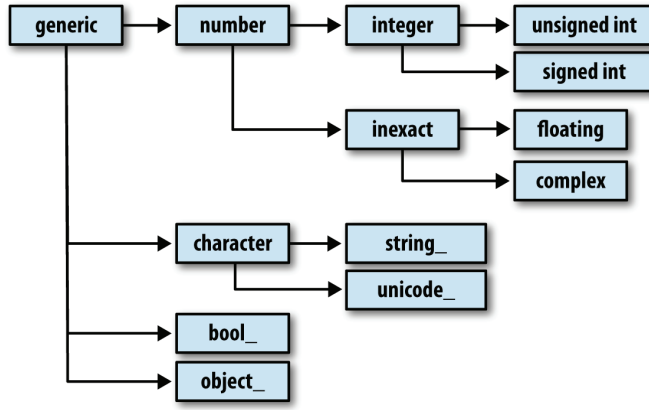


Рис. 12.2. Иерархия наследования классов dtype в NumPy

Дополнительные манипуляции с массивами

Помимо прихотливого индексирования, вырезания и формирования булевых подмножеств, существует много других способов работы с массивами. И хотя большую часть сложных задач, решаемых в ходе анализа данных, берут на себя высокоуровневые функции из библиотеки pandas, иногда возникает необходимость написать алгоритм обработки данных, которого нет в имеющихся библиотеках.

Изменение формы массива

С учетом наших знаний о массивах NumPy, не должно вызывать удивления, что изменить форму массива можно без копирования данных. Для этого следует передать кортеж с описанием новой формы методу экземпляра массива `reshape`. Например, предположим, что имеется одномерный массив, который мы хотели бы преобразовать в матрицу:

```
In [15]: arr = np.arange(8)

In [16]: arr
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7])

In [17]: arr.reshape((4, 2))
Out[17]:
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

Форму многомерного массива также можно изменить:

```
In [18]: arr.reshape((4, 2)).reshape((2, 4))
Out[18]:
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

Одно из переданных в описателе формы измерений может быть равно –1, его значение будет выведено из данных:

```
In [19]: arr = np.arange(15)
```

```
In [20]: arr.reshape((5, -1))
Out[20]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

Поскольку атрибут `shape` массива является кортежем, его также можно передать методу `reshape`:

```
In [21]: other_arr = np.ones((3, 5))

In [22]: other_arr.shape
Out[22]: (3, 5)

In [23]: arr.reshape(other_arr.shape)
Out[23]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

Обратная операция — переход от многомерного к одномерному массиву – называется *линеаризацией*:

```
In [24]: arr = np.arange(15).reshape((5, 3))
```

```
In [25]: arr
Out[25]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
In [26]: arr.ravel()
Out[26]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Метод `ravel` не создает копию данных, если без этого можно обойтись (подробнее об этом ниже). Метод `flatten` ведет себя, как `ravel`, но всегда возвращает копию данных:

```
In [27]: arr.flatten()
Out[27]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

Данные можно линеаризовать в разном порядке. Начинающим пользователям NumPy эта тема может показаться довольно сложной, поэтому ей посвящен целиком следующий подраздел.

Упорядочение элементов массива в C и в Fortran

В отличие от других сред научных расчетов, например R и MATLAB, библиотека NumPy предлагает большую гибкость в определении порядка размещения данных в памяти. По умолчанию массивы NumPy размещаются *по строкам*. Это означает, что при размещении двумерного массива в памяти соседние элементы строки находятся в соседних ячейках памяти. Альтернативой является размещение *по столбцам*, тогда (впрочем, вы уже догадались) в соседних ячейках находятся соседние элемента столбца.

По историческим причинам, порядок размещения по строкам называется порядком C, а по столбцам – порядком Fortran. В языке FORTRAN 77, на котором писали наши предки, матрицы размещались по столбцам.

Функции типа `reshape` и `ravel` принимают аргумент `order`, показывающий, в каком порядке размещать данные в массиве. Обычно задают значение `'C'` или `'F'` (о менее употребительных значениях `'A'` и `'K'` можно прочитать в документации по NumPy). Результат показан на рис. 12.3.

```
In [28]: arr = np.arange(12).reshape((3, 4))

In [29]: arr
Out[29]:
array([[ 0, 1,  2,  3],
       [ 4, 5,  6,  7],
       [ 8, 9, 10, 11]])

In [30]: arr.ravel()
Out[30]: array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])

In [31]: arr.ravel('F')
Out[31]: array([ 0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])
```

Рис. 12.3. Изменение формы массива для порядка C (по строкам) и Fortran (по столбцам)

Изменение формы массива, имеющего больше двух измерений – головоломное упражнение. Основное различие между порядком C и Fortran состоит в том, в каком порядке перебираются измерения:

- *порядок по строкам (C)*: старшие измерения обходятся *раньше* (т. е. сначала обойти ось 1, а потом переходить к оси 0);
- *порядок по столбцам (Fortran)*: старшие измерения обходятся *позже* (т. е. сначала обойти ось 0, а потом переходить к оси 1);

Конкатенация и разбиение массива

Метод `numpy.concatenate` принимает произвольную последовательность (кортеж, список и т. п.) массивов и соединяет их вместе в порядке, определяемом указанной осью.

```
In [32]: arr1 = np.array([[1, 2, 3], [4, 5, 6]])

In [33]: arr2 = np.array([[7, 8, 9], [10, 11, 12]])

In [34]: np.concatenate([arr1, arr2], axis=0)
Out[34]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [35]: np.concatenate([arr1, arr2], axis=1)
Out[35]:
array([[ 1, 2, 3,  7,  8,  9],
       [ 4, 5, 6, 10, 11, 12]])
```

Есть несколько вспомогательных функций, например `vstack` и `hstack`, для выполнения типичных операций конкатенации. Приведенные выше операции можно было бы записать и так:

```
In [36]: np.vstack((arr1, arr2))
Out[36]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

```
In [37]: np.hstack((arr1, arr2))
Out[37]:
  array([[ 1, 2, 3,  7,  8,  9],
         [ 4, 5, 6, 10, 11, 12]])
```

С другой стороны, функция `split` разбивает массив на несколько частей вдоль указанной оси:

```
In [38]: from numpy.random import randn
In [39]: arr = randn(5, 2)
```

```
In [40]: arr
Out[40]:
array([[ 0.1689, 0.3287],
       [ 0.4703, 0.8989],
       [ 0.1535, 0.0243],
       [-0.2832, 1.1536],
```

```
                                        [ 0.2707, 0.8075]])

In [41]: first, second, third = np.split(arr, [1, 3])

In [42]: first
Out[42]: array([[ 0.1689, 0.3287]])

In [43]: second                    In [44]: third
Out[43]:                           Out[44]:
array([[ 0.4703, 0.8989],          array([[-0.2832, 1.1536],
       [ 0.1535, 0.0243]])                [ 0.2707, 0.8075]])
```

Перечень всех функций, относящихся к конкатенации и разбиению, приведен в табл. 12.1, хотя некоторые из них – лишь надстройки над очень общей функцией `concatenate`.

Таблица 12.1. Функции конкатенации массива

Функция	Описание
`concatenate`	Самая общая функция – конкатенирует коллекцию массивов вдоль указанной оси
`vstack, row_stack`	Составляет массивы по строкам (вдоль оси 0)
`hstack`	Составляет массивы по столбцам (вдоль оси 1)
`column_stack`	Аналогична `hstack`, но сначала преобразует одномерные массивы с двумерные векторы по столбцам
`dstack`	Составляет массивы по в глубину (вдоль оси 2)
`split`	Разбивает массив в указанных позициях вдоль указанной оси
`hsplit / vsplit / dsplit`	Вспомогательные функции для разбиения по оси 0, 1 и 2 соответственно

Вспомогательные объекты: r_ и c_

В пространстве имен NumPy есть два специальных объекта: `r_` и `c_`, благодаря которым составление массивов можно записать более кратко:

```
In [45]: arr = np.arange(6)

In [46]: arr1 = arr.reshape((3, 2))

In [47]: arr2 = randn(3, 2)

In [48]: np.r_[arr1, arr2]          In [49]: np.c_[np.r_[arr1, arr2], arr]
Out[48]:                            Out[49]:
array([[ 0.    ,  1.    ],          array([[ 0.    ,  1.    , 0.    ],
       [ 2.    ,  3.    ],                 [ 2.    ,  3.    , 1.    ],
       [ 4.    ,  5.    ],                 [ 4.    ,  5.    , 2.    ],
       [ 0.7258, -1.5325],                 [ 0.7258, -1.5325, 3.    ],
       [-0.4696, -0.2127],                 [-0.4696, -0.2127, 4. ],
       [-0.1072, 1.2871]])                 [-0.1072, 1.2871, 5. ]])
```

С их помощью можно также преобразовывать срезы в массивы:

```
In [50]: np.c_[1:6, -10:-5]
Out[50]:
array([[ 1, -10],
       [ 2,  -9],
       [ 3,  -8],
       [ 4,  -7],
       [ 5,  -6]])
```

О том, что еще могут делать объекты `c_` и `r_`, читайте в строке документации.

Повторение элементов: функции tile и repeat

Необходимость повторять массивы при работе с NumPy возникает реже, чем в других популярных средах программирования, например MATLAB. Основная причина заключается в том, что *укладывание* (тема следующего раздела) решает эту задачу лучше.

Два основных инструмента повторения, или репликации массивов для порождения массивов большего размера – функции `repeat` и `tile`. Функция `repeat` повторяет каждый элемент массив несколько раз и создает больший массив:

```
In [51]: arr = np.arange(3)

In [52]: arr.repeat(3)
Out[52]: array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

По умолчанию, если передать целое число, то каждый элемент повторяется столько раз. Если же передать массив целых чисел, то разные элементы могут быть повторены разное число раз:

```
In [53]: arr.repeat([2, 3, 4])
Out[53]: array([0, 0, 1, 1, 1, 2, 2, 2, 2])
```

Элементы многомерных массивов повторяются вдоль указанной оси:

```
In [54]: arr = randn(2, 2)

In [55]: arr
Out[55]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])
```

```
In [56]: arr.repeat(2, axis=0)
Out[56]:
array([[ 0.7157, -0.6387],
       [ 0.7157, -0.6387],
       [ 0.3626,  0.849 ],
       [ 0.3626,  0.849 ]])
```

Отметим, что если ось не указана, то массив сначала линеаризуется, а это, скорее всего, не то, что вы хотели. Чтобы повторить разные срезы многомерного массива различное число раз, можно передать массив целых чисел:

```
In [57]: arr.repeat([2, 3], axis=0)
Out[57]:
```

```
array([[ 0.7157, -0.6387],
       [ 0.7157, -0.6387],
       [ 0.3626,  0.849 ],
       [ 0.3626,  0.849 ],
       [ 0.3626,  0.849 ]])

In [58]: arr.repeat([2, 3], axis=1)
Out[58]:
array([[ 0.7157, 0.7157, -0.6387, -0.6387, -0.6387],
       [ 0.3626, 0.3626,  0.849 ,  0.849 ,  0.849 ]])
```

Функция `tile` (замостить), с другой стороны, – просто сокращенный способ составления копий массива вдоль оси. Это можно наглядно представлять себе как «укладывание плиток»:

```
In [59]: arr
Out[59]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [60]: np.tile(arr, 2)
Out[60]:
array([[ 0.7157, -0.6387, 0.7157, -0.6387],
       [ 0.3626,  0.849 , 0.3626,  0.849 ]])
```

Второй аргумент – количество плиток; если это скаляр, то мощение производится по строкам, а не по столбцам. Но второй аргумент `tile` может быть кортежем, описывающим порядок мощения:

```
In [61]: arr
Out[61]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [62]: np.tile(arr, (2, 1))
Out[62]:
array([[ 0.7157, -0.6387],
       [ 0.3626,  0.849 ],
       [ 0.7157, -0.6387],
       [ 0.3626,  0.849 ]])

In [63]: np.tile(arr, (3, 2))
Out[63]:
array([[ 0.7157, -0.6387, 0.7157, -0.6387],
       [ 0.3626,  0.849 , 0.3626,  0.849 ],
       [ 0.7157, -0.6387, 0.7157, -0.6387],
       [ 0.3626,  0.849 , 0.3626,  0.849 ],
       [ 0.7157, -0.6387, 0.7157, -0.6387],
       [ 0.3626,  0.849 , 0.3626,  0.849 ]])
```

Эквиваленты прихотливого индексирования: функции take и put

В главе 4 описывался способ получить и установить подмножество массива с помощью *прихотливого* индексирования массивами целых чисел:

```
In [64]: arr = np.arange(10) * 100

In [65]: inds = [7, 1, 2, 6] In [66]: arr[inds]
Out[66]: array([700, 100, 200, 600])
```

Существуют и другие методы ndarray, полезные в частном случае, когда выборка производится только по одной оси:

```
In [67]: arr.take(inds)
Out[67]: array([700, 100, 200, 600])

In [68]: arr.put(inds, 42)

In [69]: arr
Out[69]: array([ 0, 42, 42, 300, 400, 500, 42, 42, 800, 900])

In [70]: arr.put(inds, [40, 41, 42, 43])

In [71]: arr
Out[71]: array([ 0, 41, 42, 300, 400, 500, 43, 40, 800, 900])
```

Чтобы использовать функцию `take` для других осей, нужно передать именованный параметр `axis`:

```
In [72]: inds = [2, 0, 2, 1]

In [73]: arr = randn(2, 4)

In [74]: arr
Out[74]:
array([[-0.8237,  2.6047, -0.4578, -1.    ],
       [ 2.3198, -1.0792,  0.518 ,  0.2527]])

In [75]: arr.take(inds, axis=1)
Out[75]:
array([[-0.4578, -0.8237, -0.4578,  2.6047],
       [ 0.518 ,  2.3198,  0.518 , -1.0792]])
```

Функция `put` не принимает аргумент `axis`, а обращается по индексу к линеаризованной версии массива (одномерному массиву, построенному в предположении, что исходный массив организован, как в C, хотя в принципе это можно было бы изменить). Следовательно, если с помощью массива индексов требуется установить элементы на других осях, то придется воспользоваться прихотливым индексированием.

На момент написания этой книги, производительность функций `take` и `put`, вообще говоря, была существенно выше, чем у эквивалентного прихотливого индексирования. Я считаю это «ошибкой» в NumPy, которая должна быть исправлена, но пока следует помнить об этом при выборке подмножеств из больших массивов с помощью массивов целых чисел.

```
In [76]: arr = randn(1000, 50)

# Случайная выборка 500 строк
In [77]: inds = np.random.permutation(1000)[:500]

In [78]: %timeit arr[inds]
```

```
1000 loops, best of 3: 356 us per loop

In [79]: %timeit arr.take(inds, axis=0)
10000 loops, best of 3: 34 us per loop
```

Укладывание

Словом «укладывание» (broadcasting) описывается способ выполнения арифметических операций над массивами разной формы. Это очень мощный механизм, но даже опытные пользователи иногда испытывают затруднения с его пониманием. Простейший пример укладывания – комбинирование скалярного значения с массивом:

```
In [80]: arr = np.arange(5)

In [81]: arr                          In [82]: arr * 4
Out[81]: array([0, 1, 2, 3, 4])       Out[82]: array([ 0, 4, 8, 12, 16])
```

Здесь мы говорим, что скалярное значение 4 уложено на все остальные элементы в результате операции умножения.

Другой пример: мы можем сделать среднее по столбцам массива равным нулю, вычтя из каждого столбца столбец, содержащий средние значения. И сделать это очень просто:

```
In [83]: arr = randn(4, 3)

In [84]: arr.mean(0)
Out[84]: array([ 0.1321, 0.552 , 0.8571])

In [85]: demeaned = arr - arr.mean(0)
In [86]: demeaned                          In [87]: demeaned.mean(0)
Out[86]:                                   Out[87]: array([ 0., -0., -0.])
array([[ 0.1718, -0.1972, -1.3669],
       [-0.1292,  1.6529, -0.3429],
       [-0.2891, -0.0435,  1.2322],
       [ 0.2465, -1.4122,  0.4776]])
```

Эта операция показана на рис. 12.4. Для приведения к нулю средних по строкам с помощью укладывания требуется проявить осторожность. К счастью, укладывание значений меньшей размерности вдоль любого измерения массива (например, вычитание средних по строкам из каждого столбца двумерного массива) возможно при соблюдении следующего правила:

Правило укладывания

Два массива совместимы по укладыванию, если для обоих *последних измерений* (т. е. отсчитываемых с конца) длины осей совпадают или хотя бы одна длина равна 1. Тогда укладывание производится по отсутствующим измерениям или по измерениям длины 1.

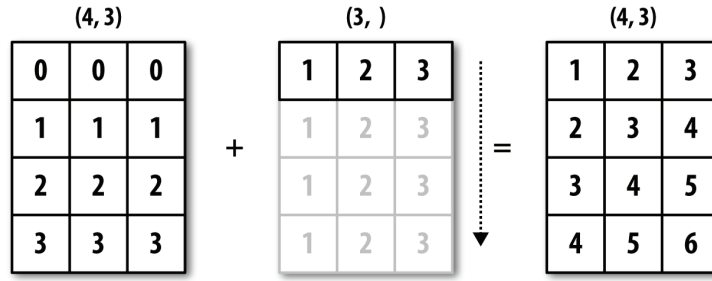


Рис. 12.4. Укладывание одномерного массива по оси 0

Даже я, опытный пользователь NumPy, иногда вынужден рисовать картинки, чтобы понять, как будет применяться правило укладывания. Вернемся к последнему примеру и предположим, что мы хотим вычесть среднее значение из каждой строки, а не из каждого столбца. Поскольку длина массива `arr.mean(0)` равна 3, он совместим по укладыванию вдоль оси 0, т. к. по последнему измерению длины осей (три) совпадают. Согласно правилу, чтобы произвести вычитание по оси 1 (т. е. вычесть среднее по строкам из каждой строки), меньший массив должен иметь форму `(4, 1)`:

```
In [88]: arr
Out[88]:
array([[ 0.3039,  0.3548, -0.5097],
       [ 0.0029,  2.2049,  0.5142],
       [-0.1571,  0.5085,  2.0893],
       [ 0.3786, -0.8602,  1.3347]])

In [89]: row_means = arr.mean(1) In [90]: row_means.reshape((4, 1))
Out[90]:
array([[ 0.0496],
       [ 0.9073],
       [ 0.8136],
       [ 0.2844]])

In [91]: demeaned = arr - row_means.reshape((4, 1))

In [92]: demeaned.mean(1)
Out[92]: array([ 0., 0., 0., 0.])
```

У вас еще мозги не кипят? Эта операция проиллюстрирована на рис. 12.5:

На рис. 12.6 приведена еще одна иллюстрация, где мы вычитаем двумерный массив из трехмерного по оси 0.

Укладывание по другим осям

Укладывание многомерных массивов может показаться еще более головоломной задачей, но на самом деле нужно только соблюдать правило. Если оно не соблюдено, то вы будет выдана ошибка вида:

```
In [93]: arr - arr.mean(1)
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-93-7b87b85a20b2> in <module>()
----> 1 arr - arr.mean(1)
ValueError: operands could not be broadcast together with shapes (4,3) (4)
```

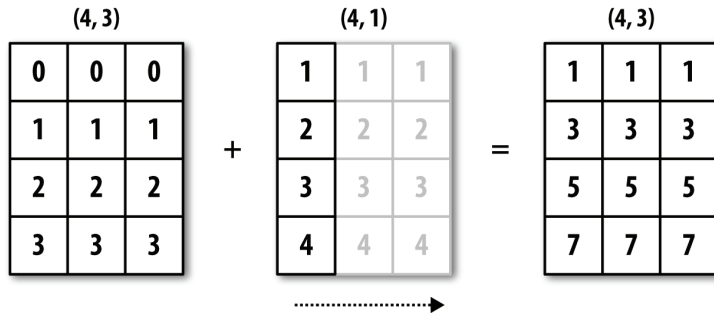


Рис. 12.5. Укладывание на двумерный массив по оси 1

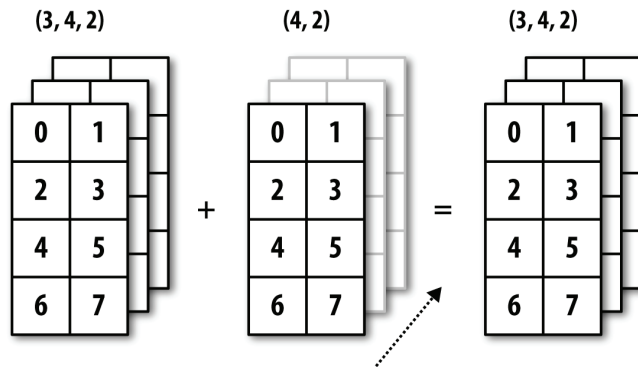


Рис. 12.6. Укладывание на трехмерный массив по оси 0

Очень часто возникает необходимость выполнить арифметическую операцию с массивом меньшей размерности по оси, отличной от 0. Согласно правилу укладывания, длина «размерности укладывания» в меньшем массиве должна быть равна 1. В примере вычитания среднего это означало, что массив средних по строкам должен иметь форму `(4, 1)`, а не `(4,)`:

```
In [94]: arr - arr.mean(1).reshape((4, 1))
Out[94]:
array([[ 0.2542,  0.3051, -0.5594],
       [-0.9044,  1.2976, -0.3931],
       [-0.9707, -0.3051,  1.2757],
       [ 0.0942, -1.1446,  1.0503]])
```

В трехмерном случае укладывание по любому из трех измерений сводится к изменению формы данных для обеспечения совместимости массивов. На рис. 12.7

наглядно показано, каковы должны быть формы для укладывания по любой оси трехмерного массива.

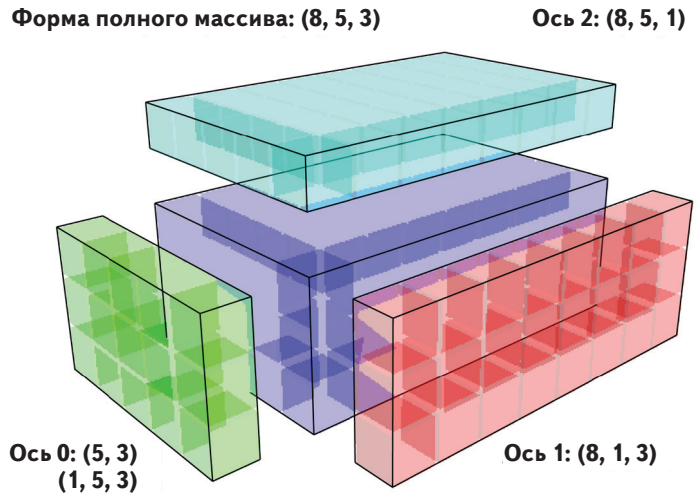


Рис. 12.7. Совместимые формы двумерного массива для укладывания в трехмерный массив

Поэтому часто приходится добавлять новую ось длины 1 специально для укладывания, особенно в обобщенных алгоритмах. Один из вариантов – использование `reshape`, но для вставки оси нужно построить кортеж, описывающий новую форму. Это утомительное занятие. Поэтому в NumPy имеется специальный синтаксис для вставки новых осей путем доступа по индексу. Чтобы вставить новую ось, мы воспользуемся специальным атрибутом `np.newaxis` и «полными» срезами:

```
In [95]: arr = np.zeros((4, 4))

In [96]: arr_3d = arr[:, np.newaxis, :] In [97]: arr_3d.shape
Out[97]: (4, 1, 4)

In [98]: arr_1d = np.random.normal(size=3)

In [99]: arr_1d[:, np.newaxis] In [100]: arr_1d[np.newaxis, :]
Out[99]:                        Out[100]: array([[-0.3899, 0.396 , -0.1852]])
array([[-0.3899],
       [ 0.396 ],
       [-0.1852]])
```

Таким образом, если имеется трехмерный массив и требуется привести его к нулевому среднему, скажем по оси 2, то нужно написать:

```
In [101]: arr = randn(3, 4, 5)

In [102]: depth_means = arr.mean(2)

In [103]: depth_means
```

```
Out[103]:
array([[ 0.1097, 0.3118, -0.5473,  0.2663],
       [ 0.1747, 0.1379,  0.1146, -0.4224],
       [ 0.0217, 0.3686, -0.0468,  1.3026]])

In [104]: demeaned = arr - depth_means[:, :, np.newaxis]
In [105]: demeaned.mean(2)
Out[105]:
array([[ 0.,  0., -0., 0.],
       [ 0., -0., -0., 0.],
       [-0., -0.,  0., 0.]])
```

Если вы окончательно запутались, не переживайте! Мудрость достигается практикой!

Кто-то может задаться вопросом, а нет ли способа обобщить вычитание среднего вдоль оси, не жертвуя производительностью. Есть, но потребуется попотеть с индексированием:

```
def demean_axis(arr, axis=0):
  means = arr.mean(axis)
  # Это обобщает операции вида [:, :, np.newaxis] на N измерений
  indexer = [slice(None)] * arr.ndim
  indexer[axis] = np.newaxis
  return arr - means[indexer]
```

Установка элементов массива с помощью укладывания

То же правило укладывания, что управляет арифметическими операциями, применимо и к установке значений элементов с помощью доступа по индексу. В простейшем случае это выглядит так:

```
In [106]: arr = np.zeros((4, 3))

In [107]: arr[:] = 5
```

```
In [108]: arr
Out[108]:
array([[ 5., 5., 5.],
       [ 5., 5., 5.],
       [ 5., 5., 5.],
       [ 5., 5., 5.]])
```

Однако если имеется одномерный массив значений, который требуется записать в столбцы массива, то можно сделать и это – при условии совместимости формы:

```
In [109]: col = np.array([1.28, -0.42, 0.44, 1.6])

In [110]: arr[:] = col[:, np.newaxis]
```

```
In [111]: arr
Out[111]:
array([[ 1.28,  1.28,  1.28],
       [-0.42, -0.42, -0.42],
```

```
                                        [ 0.44,  0.44,  0.44],
                                        [ 1.6 ,  1.6 ,  1.6 ]])

In [112]: arr[:2] = [[-1.37], [0.509]]   In [113]: arr
                                         Out[113]:
                                         array([[-1.37 , -1.37 , -1.37 ],
                                                [ 0.509,  0.509,  0.509],
                                                [ 0.44 ,  0.44 ,  0.44 ],
                                                [ 1.6  ,  1.6  ,  1.6  ]])
```

Дополнительные способы использования универсальных функций

Многие пользователи NumPy используют универсальные функции только ради быстрого выполнения поэлементных операций, однако у них есть и другие возможности, которые иногда позволют кратко записать код без циклов.

Методы экземпляра u-функций

Любая бинарная u-функция в NumPy имеет специальные методы для выполнения некоторых видов векторных операций. Все они перечислены в табл. 12.2, но я приведу и несколько конкретных примеров для иллюстрации.

Метод `reduce` принимает массив и агрегирует его, возможно вдоль указанной оси, выполняя последовательность бинарных операций. Вот, например, как можно с помощью `np.add.reduce` просуммировать элементы массива:

```
In [114]: arr = np.arange(10)

In [115]: np.add.reduce(arr)
Out[115]: 45

In [116]: arr.sum()
Out[116]: 45
```

Начальное значение (для `add` оно равно 0) зависит от u-функции. Если задана ось, то редукция производится вдоль этой оси. Это позволяет давать краткие ответы на некоторые вопросы. В качестве не столь тривиального примера воспользуемся методом `np.logical_and`, чтобы проверить, отсортированы ли значения в каждой строке массива:

```
In [118]: arr = randn(5, 5)

In [119]: arr[::2].sort(1) # sort a few rows

In [120]: arr[:, :-1] < arr[:, 1:]
Out[120]:
array([[ True,  True,  True,  True],
```

```
       [False,  True, False, False],
       [ True,  True,  True,  True],
       [ True, False,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

In [121]: np.logical_and.reduce(arr[:, :-1] < arr[:, 1:], axis=1)
Out[121]: array([ True, False, True, False, True], dtype=bool)
```

Разумеется, `logical_and.reduce` эквивалентно методу `all`.

Метод `accumulate` соотносится с `reduce`, как `cumsum` с `sum`. Он порождает массив того же размера, содержащий промежуточные «аккумулированные» значения:

```
In [122]: arr = np.arange(15).reshape((3, 5))

In [123]: np.add.accumulate(arr, axis=1)
Out[123]:
array([[ 0,  1,  3,  6, 10],
       [ 5, 11, 18, 26, 35],
       [10, 21, 33, 46, 60]])
```

Метод `outer` вычисляет прямое произведение двух массивов:

```
In [124]: arr = np.arange(3).repeat([1, 2, 2])

In [125]: arr
Out[125]: array([0, 1, 1, 2, 2])

In [126]: np.multiply.outer(arr, np.arange(5))
Out[126]:
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 2, 4, 6, 8],
       [0, 2, 4, 6, 8]])
```

Размерность массива, возвращенного `outer`, равна сумме размерностей его параметров:

```
In [127]: result = np.subtract.outer(randn(3, 4), randn(5))

In [128]: result.shape
Out[128]: (3, 4, 5)
```

Последний метод, `reduceat`, выполняет «локальную редукцию», т. е. по существу операцию `groupby`, в которой агрегируется сразу несколько срезов массива. Хотя он не такой гибкий, как механизм GroupBy в pandas, при некоторых обстоятельствах оказывается очень быстрым и эффективным. Он принимает «границы интервалов», описывающие, как разбивать и агрегировать значения:

```
In [129]: arr = np.arange(10)

In [130]: np.add.reduceat(arr, [0, 5, 8])
Out[130]: array([10, 18, 17])
```

На выходе получаются результаты редукции (в данном случае суммирования) по срезам `arr[0:5]`, `arr[5:8]` и `arr[8:]`. Как и другие методы, `reduceat` принимает необязательный аргумент `axis`:

```
In [131]: arr = np.multiply.outer(np.arange(4), np.arange(5))

In [132]: arr
Out[132]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12]])

In [133]: np.add.reduceat(arr, [0, 2, 4], axis=1)
Out[133]:
array([[ 0,  0,  0],
       [ 1,  5,  4],
       [ 2, 10,  8],
       [ 3, 15, 12]])
```

Таблица 12.2. Методы u-функций

Метод	Описание
`reduce(x)`	Агрегирует значения путем последовательного применения операции
`accumulate(x)`	Агрегирует значения, сохраняя все промежуточные агрегаты
`reduceat(x, bins)`	«Локальная» редукция, или «group by». Редуцирует соседние срезы данных и порождает массив агрегатов
`outer(x, y)`	Применяет операцию ко всем парам элементов `x` и `y`. Результирующий массив имеет форму `x.shape + y.shape`

Пользовательские u-функции

Существует два механизма создания собственных функций с такой же семантикой, как у u-функций. Метод `numpy.frompyfunc` принимает функцию Python и спецификацию количества входов и выходов. Например, простую функцию, выполняющую поэлементное сложение, можно было бы описать так:

```
In [134]: def add_elements(x, y):
   .....:     return x + y

In [135]: add_them = np.frompyfunc(add_elements, 2, 1)

In [136]: add_them(np.arange(8), np.arange(8))
Out[136]: array([0, 2, 4, 6, 8, 10, 12, 14], dtype=object)
```

Функции, созданные методом `frompyfunc`, всегда возвращают массивы объектов Python, что не очень удобно. По счастью, есть альтернативный, хотя и не столь функционально богатый метод `numpy.vectorize`, который умеет лучше выводить типы:

```
In [137]: add_them = np.vectorize(add_elements, otypes=[np.float64])

In [138]: add_them(np.arange(8), np.arange(8))
Out[138]: array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14.])
```

Оба метода позволяют создавать аналоги u-функций, которые, правда, работают очень медленно, потому что должны вызывать функцию Python для вычисления каждого элемента, а это далеко не так эффективно, как циклы в написанных на C универсальных функциях NumPy:

```
In [139]: arr = randn(10000)

In [140]: %timeit add_them(arr, arr)
100 loops, best of 3: 2.12 ms per loop

In [141]: %timeit np.add(arr, arr)
100000 loops, best of 3: 11.6 us per loop
```

В настоящее время в сообществе научного Python разрабатывается несколько проектов, которые должны упростить определение новых u-функций с примерно такой же производительностью, как у встроенных.

Структурные массивы

Вы, наверное, обратили внимание, что все рассмотренные до сих пор примеры ndarray были контейнерами *однородных* данных, т. е. блоками памяти, в которых каждый элемент занимает одно и то же количество байтов, определяемое типом данных dtype. Создается впечатление, что представить в виде массива неоднородные данные, как в таблице, невозможно. *Структурный* массив – это объект ndarray, в котором каждый элемент можно рассматривать как аналог *структуры* (struct) в языке C (отсюда и название «структурный») или строки в таблице SQL, содержащий несколько именованных полей:

```
In [142]: dtype = [('x', np.float64), ('y', np.int32)]

In [143]: sarr = np.array([(1.5, 6), (np.pi, -2)], dtype=dtype)

In [144]: sarr
Out[144]:
array([(1.5, 6), (3.141592653589793, -2)],
      dtype=[('x', '<f8'), ('y', '<i4')])
```

Существует несколько способов задать структурный dtype (см. документацию по NumPy в сети). Наиболее распространенный – с помощью списка кортежей вида `(field_name, field_data_type)`. Теперь элементами массива являются кортежеподобные объекты, к элементам которых можно обращаться как к словарю:

```
In [145]: sarr[0]
Out[145]: (1.5, 6)

In [146]: sarr[0]['y']
Out[146]: 6
```

Имена полей хранятся в атрибуте `dtype.names`. При доступе к полю структурного массива возвращается шаговое представление данных, т. е. копирования не происходит:

```
In [147]: sarr['x']
Out[147]: array([ 1.5 , 3.1416])
```

Вложенные типы данных и многомерные поля

При описании структурного dtype можно факультативно передать форму (в виде целого числа или кортежа):

```
In [148]: dtype = [('x', np.int64, 3), ('y', np.int32)]

In [149]: arr = np.zeros(4, dtype=dtype)

In [150]: arr
Out[150]:
array([([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0), ([0, 0, 0], 0)],
      dtype=[('x', '<i8', (3,)), ('y', '<i4')])
```

В данном случае поле `x` в каждой записи ссылается на массив длиной 3:

```
In [151]: arr[0]['x']
Out[151]: array([0, 0, 0])
```

При этом результатом операции `arr['x']` является двумерный массив, а не одномерный, как в предыдущих примерах:

```
In [152]: arr['x']
Out[152]:
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Это позволяет представлять более сложные вложенные структуры в виде одного блока памяти в массиве. Но если dtype может быть произвольно сложным, то почему бы и не вложенным? Вот простой пример:

```
In [153]: dtype = [('x', [('a', 'f8'), ('b', 'f4')]), ('y', np.int32)]

In [154]: data = np.array([((1, 2), 5), ((3, 4), 6)], dtype=dtype)

In [155]: data['x']
Out[155]:
array([(1.0, 2.0), (3.0, 4.0)],
      dtype=[('a', '<f8'), ('b', '<f4')])

In [156]: data['y']
Out[156]: array([5, 6], dtype=int32)

In [157]: data['x']['a']
Out[157]: array([ 1., 3.])
```

Как видите, поля переменной формы и вложенные записи – очень мощное средство, которое может оказаться весьма полезным в некоторых ситуациях. Объект DataFrame из библиотеки pandas не поддерживает этот механизм напрямую, хотя иерархическое индексирование в чем-то похоже.

Зачем нужны структурные массивы?

По сравнению, скажем, с объектом DataFrame из pandas, структурные массивы NumPy – средство относительно низкого уровня. Они позволяют интерпретировать блок памяти как табличную структуру с вложенными столбцами произвольной сложности. Поскольку каждый элемент представлен в памяти фиксированным количеством байтов, структурный массив дает очень быстрый и эффективный способ записи данных на диск и чтения с диска (в том числе в файлы, спроецированные на память, о чем речь пойдет ниже), передачи по сети и прочих операций такого рода.

Еще одно распространенное применение структурных массивов связано со стандартным способом сериализации данных в C и C++, часто встречающимся в унаследованных системах; данные выводятся в файл в виде потока байтов с фиксированной длиной записи. Коль скоро известен формат файла (размер каждой записи, порядок байтов и тип данных каждого элемента), данные можно прочитать в память методом `np.fromfile`. Подобные специализированные применения выходят за рамки этой книги, но знать об их существовании полезно.

Манипуляции со структурными массивами: numpy.lib.recfunctions

Хотя набор средств для работы со структурными массивами не так богат, как для объектов DataFrames, в модуле NumPy `numpy.lib.recfunctions` имеются некоторые полезные функции для добавления и удаления полей и выполнения простых операций соединения. Нужно только помнить, что при работе с этими функциями обычно необходимо создавать новый массив, чтобы изменения dtype (скажем, добавление или удаление столбца) вступили в силу. Изучение этих функций я оставляю интересующимся читателям, поскольку в книге они не используются.

Еще о сортировке

Как и у встроенных списков Python, метод `sort` объекта производит сортировку *на месте*, т. е. массив переупорядочивается без порождения нового массива:

```
In [158]: arr = randn(6)

In [159]: arr.sort()

In [160]: arr
Out[160]: array([-1.082 , 0.3759, 0.8014, 1.1397, 1.2888, 1.8413])
```

Сортируя на месте, не забывайте, что если сортируемый массив — представление другого массива ndarray, то модифицируется исходный массив:

```
In [161]: arr = randn(3, 5)

In [162]: arr
Out[162]:
array([[-0.3318, -1.4711, 0.8705, -0.0847, -1.1329],
       [-1.0111, -0.3436, 2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424, 0.8548,  1.038 , -0.329 ]])

In [163]: arr[:, 0].sort() # Sort first column values in-place

In [164]: arr
Out[164]:
array([[-1.0111, -1.4711, 0.8705, -0.0847, -1.1329],
       [-0.3318, -0.3436, 2.1714,  0.1234, -0.0189],
       [ 0.1773,  0.7424, 0.8548,  1.038 , -0.329 ]])
```

С другой стороны, функция `numpy.sort` создает отсортированную копию массива, принимая те же самые аргументы (в частности, `kind`, о котором будет сказано ниже), что и метод `ndarray.sort`:

```
In [165]: arr = randn(5)

In [166]: arr
Out[166]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])

In [167]: np.sort(arr)
Out[167]: array([-2.0051, -1.1181, -1.0614, -0.2415, 0.7379])

In [168]: arr
Out[168]: array([-1.1181, -0.2415, -2.0051, 0.7379, -1.0614])
```

Все методы сортировки принимают аргумент `axis`, что позволяет независимо сортировать участки массива вдоль указанной оси:

```
In [169]: arr = randn(3, 5)

In [170]: arr
Out[170]:
array([[ 0.5955, -0.2682,  1.3389, -0.1872,  0.9111],
       [-0.3215,  1.0054, -0.5168,  1.1925, -0.1989],
       [ 0.3969, -1.7638,  0.6071, -0.2222, -0.2171]])

In [171]: arr.sort(axis=1)

In [172]: arr
Out[172]:
array([[-0.2682, -0.1872,  0.5955, 0.9111, 1.3389],
       [-0.5168, -0.3215, -0.1989, 1.0054, 1.1925],
       [-1.7638, -0.2222, -0.2171, 0.3969, 0.6071]])
```

Вероятно, вы обратили внимание, что ни у одного метода нет параметра, который задавал бы сортировку в порядке убывания. Но это не составляет проблемы,

потому что вырезание массива порождает представления, для чего не требуется копирование или еще какие-то вычисления. Многие пользователи Python знают, что если `values` – список, то `values[::-1]` возвращает его в обратном порядке. То же справедливо и для объектов ndarray:

```
In [173]: arr[:, ::-1]
Out[173]:
array([[ 1.3389, 0.9111,  0.5955, -0.1872, -0.2682],
       [ 1.1925, 1.0054, -0.1989, -0.3215, -0.5168],
       [ 0.6071, 0.3969, -0.2171, -0.2222, -1.7638]])
```

Косвенная сортировка: методы argsort и lexsort

В ходе анализа данных очень часто возникает необходимость переупорядочить набор данных по одному или нескольким ключам. Например, отсортировать таблицу, содержащую данные о студентах, сначала по фамилии, а потом по имени. Это пример *косвенной* сортировки, и, если вы читали главы, относящиеся к библиотеке pandas, то видели много других примеров более высокого уровня. Имея один или несколько ключей (массив или несколько массивов значений), мы хотим получить массив целочисленных *индексов* (буду называть их просто *индексаторами*), который говорит, как переупорядочить данные в нужном порядке сортировки. Для этого существуют два основных метода: `argsort` и `numpy.lexsort`. Вот тривиальный пример:

```
In [174]: values = np.array([5, 0, 1, 3, 2])

In [175]: indexer = values.argsort()

In [176]: indexer
Out[176]: array([1, 2, 4, 3, 0])

In [177]: values[indexer]
Out[177]: array([0, 1, 2, 3, 5])
```

А в следующем, чуть более сложном примере двумерный массив переупорядочивается по первой строке:

```
In [178]: arr = randn(3, 5)

In [179]: arr[0] = values

In [180]: arr
Out[180]:
array([[ 5.    ,  0.    , 1.    ,  3.    , 2. ],
       [-0.3636, -0.1378, 2.1777, -0.4728, 0.8356],
       [-0.2089,  0.2316, 0.728 , -1.3918, 1.9956]])

In [181]: arr[:, arr[0].argsort()]
Out[181]:
array([[ 0.    ,    1. ,    2. ,     3. , 5.    ],
       [-0.1378, 2.1777, 0.8356, -0.4728, -0.3636],
       [ 0.2316, 0.728 , 1.9956, -1.3918, -0.2089]])
```

Метод `lexsort` аналогичен `argsort`, но выполняет косвенную *лексикографическую* сортировку по нескольким массивам ключей. Пусть требуется отсортировать данные, идентифицируемые именем и фамилией:

```
In [182]: first_name = np.array(['Bob', 'Jane', 'Steve', 'Bill', 'Barbara'])

In [183]: last_name = np.array(['Jones', 'Arnold', 'Arnold', 'Jones',
'Walters'])

In [184]: sorter = np.lexsort((first_name, last_name))

In [185]: zip(last_name[sorter], first_name[sorter])
Out[185]:
[('Arnold', 'Jane'),
 ('Arnold', 'Steve'),
 ('Jones', 'Bill'),
 ('Jones', 'Bob'),
 ('Walters', 'Barbara')]
```

При первом использовании метод `lexsort` может вызвать недоумение, потому что первым для сортировки используется ключ, указанный в *последнем* массиве. Как видите, ключ `last_name` использовался раньше, чем `first_name`. В библиотеке pandas методы `sort_index` объектов Series и DataFrame, а также метод `order` объекта Series реализованы с помощью вариантов этих функций (в которые добавлен учет отсутствующих значений).

Альтернативные алгоритмы сортировки

Устойчивый алгоритм сортировки сохраняет относительные позиции равных элементов. Это особенно важно при косвенной сортировке, когда относительный порядок имеет значение:

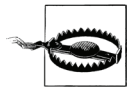
```
In [186]: values = np.array(['2:first', '2:second', '1:first', '1:second',
'1:third'])

In [187]: key = np.array([2, 2, 1, 1, 1])

In [188]: indexer = key.argsort(kind='mergesort')

In [189]: indexer
Out[189]: array([2, 3, 4, 0, 1])

In [190]: values.take(indexer)
Out[190]:
array(['1:first', '1:second', '1:third', '2:first', '2:second'],
      dtype='|S8')
```

Единственный имеющийся устойчивый алгоритм сортировки с гарантированным временем работы O(n log n) – *mergesort*, но его производительность в среднем хуже, чем у алгоритма *quicksort*. В табл. 12.3 перечислены имеющиеся алгоритмы, их сравнительное быстродействие и гарантированная производительность. Боль-

шинству пользователей эта информация не особенно интересна, но знать о ее существовании стоит.

Таблица 12.3. Алгоритмы сортировки массива

Алгоритм	Быстродействие	Устойчивый	Рабочая память	В худшем случае
`'quicksort'`	1	Нет	0	$O(n^2)$
`'mergesort'`	2	Да	n / 2	O(n log n)
`'heapsort'`	3	Нет	0	O(n log n)



На момент написания этой книги для массивов объектов Python (`dtype=object`) был доступен только алгоритм сортировки quicksort. Это означает, что в тех случаях, когда требуется устойчивая сортировка, нужно как-то обходиться без использования объектов Python.

Метод numpy.searchsorted: поиск элементов в отсортированном массиве

Метод массива `searchsorted` производит двоичный поиск в отсортированном массиве и возвращает место, в которое нужно было бы вставить значение, чтобы массив оставался отсортированным:

```
In [191]: arr = np.array([0, 1, 7, 12, 15])

In [192]: arr.searchsorted(9)
Out[192]: 3
```

Как и следовало ожидать, можно передать также массив значений и получить в ответ массив индексов:

```
In [193]: arr.searchsorted([0, 8, 11, 16])
Out[193]: array([0, 3, 3, 5])
```

Вы, наверное, заметили, что `searchsorted` вернул индекс `0` для элемента `0`. Это объясняется тем, что по умолчанию возвращается индекс левого из группы элементов с одинаковыми значениями:

```
In [194]: arr = np.array([0, 0, 0, 1, 1, 1, 1])

In [195]: arr.searchsorted([0, 1])
Out[195]: array([0, 3])

In [196]: arr.searchsorted([0, 1], side='right')
Out[196]: array([3, 7])
```

Чтобы проиллюстрировать еще одно применение метода `searchsorted`, предположим, что имеется массив значений между 0 и 10 000 и отдельный массив «границ интервалов», который мы хотим использовать для распределения данных по интервалам:

```
In [197]: data = np.floor(np.random.uniform(0, 10000, size=50))

In [198]: bins = np.array([0, 100, 1000, 5000, 10000])

In [199]: data

Out[199]:
array([ 8304., 4181., 9352., 4907., 3250., 8546., 2673., 6152.,
        2774., 5130., 9553., 4997., 1794., 9688.,  426., 1612.,
         651., 8653., 1695., 4764., 1052., 4836., 8020., 3479.,
        1513., 5872., 8992., 7656., 4764., 5383., 2319., 4280.,
        4150., 8601., 3946., 9904., 7286., 9969., 6032., 4574.,
        8480., 4298., 2708., 7358., 6439., 7916., 3899., 9182.,
         871., 7973.])
```

Чтобы теперь для каждой точки узнать, какому интервалу она принадлежит (считая, что 1 означает интервал `[0, 100)`), мы можем воспользоваться методом `searchsorted`:

```
In [200]: labels = bins.searchsorted(data)
In [201]: labels
Out[201]:
array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4,
       3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4,
       3, 4, 2, 4])
```

В сочетании с методом `groupby` из библиотеки pandas этого достаточно, чтобы распределить данные по интервалам:

```
In [202]: Series(data).groupby(labels).mean()
Out[202]:
2    649.333333
3   3411.521739
4   7935.041667
```

Отметим, что в NumPy есть функция `digitize`, которая также вычисляет, в какой интервал попадает каждая точка:

```
In [203]: np.digitize(data, bins)
Out[203]:
array([4, 3, 4, 3, 3, 4, 3, 4, 3, 4, 4, 3, 3, 4, 2, 3, 2, 4, 3, 3, 3, 3, 4,
       3, 3, 4, 4, 4, 3, 4, 3, 3, 3, 4, 3, 4, 4, 4, 4, 3, 4, 3, 3, 4, 4, 4,
       3, 4, 2, 4])
```

Класс matrix в NumPy

По сравнению с другими языками для работы с матрицами типа MATLAB, Julia и GAUSS, синтаксис операций линейной алгебры в NumPy часто оказывается чрезмерно многословным. Одна из причин заключается в том, что для умножения матриц требуется использовать метод `numpy.dot`. Кроме того, семантика индексирования в NumPy отличается, что иногда затрудняет перенос кода на Python. Выборка одной строки (например, `X[1, :]`) или одного столбца (например,

`X[:, 1]`) из двумерного массива дает одномерный массив, а не двумерный, как, скажем, в MATLAB.

```
In [204]: X = np.array([[ 8.82768214, 3.82222409, -1.14276475, 2.04411587],
   .....:               [ 3.82222409, 6.75272284,  0.83909108, 2.08293758],
   .....:               [-1.14276475, 0.83909108,  5.01690521, 0.79573241],
   .....:               [ 2.04411587, 2.08293758,  0.79573241, 6.24095859]])

In [205]: X[:, 0] # одномерный
Out[205]: array([ 8.8277, 3.8222, -1.1428, 2.0441])

In [206]: y = X[:, :1] # двумерный за счет вырезания

In [207]: X
Out[207]:
array([[ 8.8277, 3.8222, -1.1428, 2.0441],
       [ 3.8222, 6.7527,  0.8391, 2.0829],
       [-1.1428, 0.8391,  5.0169, 0.7957],
       [ 2.0441, 2.0829,  0.7957, 6.241 ]])

In [208]: y
Out[208]:
array([[ 8.8277],
       [ 3.8222],
       [-1.1428],
       [ 2.0441]])
```

В данном случае произведение y^TXy можно было бы записать так:

```
In [209]: np.dot(y.T, np.dot(X, y))
Out[209]: array([[ 1195.468]])
```

Чтобы упростить написание кода с большим количеством операций над матрицами, в NumPy имеется класс `matrix`, в котором поведение индексирования модифицировано так, чтобы оно больше напоминало MATLAB: выборка одиночных строк и столбцов возвращает двумерный массива, а оператор `*` обозначает умножение матриц. Вышеупомянутая операция с помощью `numpy.matrix` записывается так:

```
In [210]: Xm = np.matrix(X)

In [211]: ym = Xm[:, 0]

In [212]: Xm
Out[212]:
matrix([[ 8.8277, 3.8222, -1.1428, 2.0441],
        [ 3.8222, 6.7527,  0.8391, 2.0829],
        [-1.1428, 0.8391,  5.0169, 0.7957],
        [ 2.0441, 2.0829,  0.7957, 6.241 ]])

In [213]: ym
Out[213]:
matrix([[ 8.8277],
```

```
        [ 3.8222],
        [-1.1428],
        [ 2.0441]])

In [214]: ym.T * Xm * ym
Out[214]: matrix([[ 1195.468]])
```

В классе `matrix` имеется также специальный атрибут `I`, который возвращает обратную матрицу:

```
In [215]: Xm.I * X
Out[215]:
matrix([[ 1., -0., -0., -0.],
        [ 0.,  1.,  0.,  0.],
        [ 0.,  0.,  1.,  0.],
        [ 0.,  0.,  0.,  1.]])
```

Я не рекомендую использовать класс `numpy.matrix` как замену обычным объектам ndarray, потому что используется он сравнительно редко. В отдельных функциях, где много операций линейной алгебры, иногда полезно преобразовать аргумент функции к типу `matrix`, а перед возвратом преобразовать результат в массив методом `np.asarray` (который не копирует никаких данных).

Дополнительные сведения о вводе-выводе массивов

В главе 4 мы познакомились с методами `np.save` и `np.load` для хранения массивов в двоичном формате на диске. Но есть и целый ряд дополнительных возможностей на случай, когда нужно что-то более сложное. В частности, файлы, спроецированные на память, позволяют работать с наборами данных, не умещающимися в оперативной памяти.

Файлы, спроецированные на память

Проецирование файла на память – метод, позволяющий рассматривать потенциально очень большой набор данных на диске как массив в памяти. В NumPy объект `memmap` реализован по аналогии с ndarray, он позволяет читать и записывать небольшие сегменты большого файла, не загружая в память весь массив. Кроме того, у объекта `memmap` точно такие же методы, как у массива в памяти, поэтому его можно подставить во многие алгоритмы, ожидающие получить ndarray. Для создания объекта `memmap` служит функция `np.memmap`, которой передается путь к файлу, dtype, форма и режим открытия файла:

```
In [216]: mmap = np.memmap('mymmap', dtype='float64', mode='w+', shape=(10000, 10000))

In [217]: mmap
Out[217]:
```

```
memmap([[ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        ...,
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.],
        [ 0., 0., 0., ..., 0., 0., 0.]])
```

При вырезании из `memmap` возвращается представление данных на диске:

```
In [218]: section = mmap[:5]
```

Если присвоить такому срезу значения, то они буферизуются в памяти (в виде файлового объекта Python) и могут быть записаны на диск позже методом `flush`:

```
In [219]: section[:] = np.random.randn(5, 10000)

In [220]: mmap.flush()

In [221]: mmap
Out[221]:
memmap([[-0.1614, -0.1768,  0.422 , ..., -0.2195, -0.1256, -0.4012],
        [ 0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782,  1.3208],
        [-0.6875,  1.6901, -0.7444, ..., -1.4218, -0.0509,  1.2224],
        ...,
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ]])

In [222]: del mmap
```

Сброс на диск всех изменений автоматически происходит и тогда, когда объект `memmap` выходит из области видимости и передается сборщику мусора. При *открытии существующего файла* все равно необходимо указывать dtype и форму, поскольку файл на диске – это просто блок двоичных данных без каких-либо метаданных:

```
In [223]: mmap = np.memmap('mymmap', dtype='float64', shape=(10000, 10000))

In [224]: mmap
Out[224]:
memmap([[-0.1614, -0.1768,  0.422 , ..., -0.2195, -0.1256, -0.4012],
        [ 0.4898, -2.2219, -0.7684, ..., -2.3517, -1.0782,  1.3208],
        [-0.6875,  1.6901, -0.7444, ..., -1.4218, -0.0509,  1.2224],
        ...,
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ],
        [ 0. , 0. , 0. , ..., 0. , 0. , 0. ]])
```

Поскольку проекция на память – это просто объект ndarray на диске, ничто не мешает использовать структурный dtype, как описано выше.

HDF5 и другие варианты хранения массива

PyTables и h5py – написанные на Python проекты, в которых реализован ориентированный на NumPy интерфейс для хранения массива в эффективном, допускающем сжатие формате HDF5 (HDF означает *hierarchical data format* – иерархический формат данных). В формате HDF5 можно без опаски хранить сотни гигабайтов и даже терабайты данных. К сожалению, рассмотрение этих библиотек выходит за рамки книги.

PyTables предлагает развитый механизм для работы со структурными массивами, располагающий средствами формулирования сложных запросов и построения индексов по столбцам для ускорения поиска. Это очень напоминает средства индексирования таблиц в реляционных базах данных.

Замечание о производительности

Добиться хорошей производительности программы, написанной с использованием NumPy, обычно нетрудно, поскольку операции над массивами как правило заменяют медленные по сравнению с ними циклы на чистом Python. Ниже перечислены некоторые моменты, о которых стоит помнить.

- Преобразуйте циклы и условную логику Python с операции с массивами и и булевыми массивами.
- Всюду, где только можно, применяйте укладывание.
- Избегайте копирования данных с помощью представлений массивов (вырезание).
- Используйте u-функции и их методы.

Если с помощью одних лишь средств NumPy все же никак не удается добиться требуемой производительности, то, возможно, имеет смысл написать часть кода на C, Fortran и особенно на Cython (подробнее об этом ниже). Лично я очень активно использую Cython (*http://cython.org*) в собственной работе как простой способ получить производительность, сравнимую с C, затратив минимум усилий.

Важность непрерывной памяти

Хотя полное рассмотрение заявленной темы выходит за рамки этой книги, в некоторых приложениях расположение массива в памяти может оказать существенное влияние на скорость вычислений. Отчасти это связано с иерархией процессорных кэшей; операции, в которых осуществляется доступ к соседним адресам в памяти (например, суммирование по строкам в массиве, организованном, как в C) обычно выполняются быстрее всего, потому что подсистема памяти буферизует соответствующие участки в сверхбыстром кэше уровня L1 или L2. Кроме того, некоторые ветви написанного на C кода NumPy оптимизированы в расчете на непрерывный случай, когда шагового доступа можно избежать.

Говоря о *непрерывной* организации памяти, мы имеем в виду, что элементы массива хранятся в памяти в том порядке, в котором видны в массиве, организован-

ном по столбцам (как в Fortran) или по строкам (как в C). По умолчанию массивы в NumPy создаются *C-непрерывными*. О массиве, хранящемся по столбцам, например транспонированном C-непрерывном массиве, говорят, что он Fortran-непрерывный. Эти свойства можно явно опросить с помощью атрибута `flags` объекта `ndarray`:

```
In [227]: arr_c = np.ones((1000, 1000), order='C')

In [228]: arr_f = np.ones((1000, 1000), order='F')

In [229]: arr_c.flags
Out[229]:
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

In [230]: arr_f.flags
Out[230]:
C_CONTIGUOUS : False
F_CONTIGUOUS : True
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False

In [231]: arr_f.flags.f_contiguous
Out[231]: True
```

В данном случае суммирование строк массива теоретически должно быть быстрее для `arr_c`, чем для `arr_f`, поскольку строки хранятся в памяти непрерывно. Я проверил это с помощью функции `%timeit` в IPython:

```
In [232]: %timeit arr_c.sum(1)
1000 loops, best of 3: 1.33 ms per loop

In [233]: %timeit arr_f.sum(1)
100 loops, best of 3: 8.75 ms per loop
```

Часто именно в этом направлении имеет смысл прикладывать усилия, стремясь выжать всю возможную производительность из NumPy. Если массив размещен в памяти не так, как нужно, можно скопировать его методом `copy`, передав параметр `'C'` или `'F'`:

```
In [234]: arr_f.copy('C').flags
Out[234]:
C_CONTIGUOUS : True
F_CONTIGUOUS : False
OWNDATA : True
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

При построении представления массива помните, что гарантии непрерывности результата никто не дает:

```
In [235]: arr_c[:50].flags.contiguous
Out[235]: True

In [236]: arr_c[:, :50].flags
Out[236]:
  C_CONTIGUOUS : False
```

```
F_CONTIGUOUS : False
OWNDATA : False
WRITEABLE : True
ALIGNED : True
UPDATEIFCOPY : False
```

Другие возможности ускорения: Cython, f2py, C

За последние годы проект Cython (*http://cython.org*) стал излюбленным инструментом многих разработчиков, пишущих на Python код научных приложений, который должен взаимодействовать с библиотеками на C или C++. Можно считать, что Cython – это Python со статическими типами и возможностью вставлять написанные на C функции в код на языке, похожем на Python. Например, на Cython простая функция суммирования элементов одномерного массива может выглядеть так:

```
from numpy cimport ndarray, float64_t

def sum_elements(ndarray[float64_t] arr):
  cdef Py_ssize_t i, n = len(arr)
  cdef float64_t result = 0

  for i in range(n):
     result += arr[i]

  return result
```

Cython транслирует этот код C, а затем компилирует сгенерированный C-код, создавая расширение Python. Cython – привлекательная возможность повысить производительность вычислений, потому что писать на нем код лишь чуть дольше, чем на чистом Python, и к тому же он тесно интегрирован с NumPy. Типичная последовательность операций выглядит так: добиться работоспособности алгоритма на Python, а затем перевести код на Cython, добавив объявления типов и кое-какие ухищрения. Дополнительные сведения смотрите в документации проекта.

Из других вариантов написания высокопроизводительного кода с использованием NumPy можно упомянуть f2py, генератор оберток для кода на Fortran 77 и 90, а также написание расширений на чистом C.

ПРИЛОЖЕНИЕ. Основы языка Python

Знание – сокровищница, а практика – ключ к ней.
Томас Фуллер

Меня часто просят назвать хорошие источники для изучения Python в применении к приложениям для обработки данных. И хотя существует немало отличных книг по языку Python, я обычно не тороплюсь рекомендовать некоторые из них, потому что они рассчитаны на широкую аудиторию, а не специально на тех, кому нужно загружать наборы данных, делать расчеты и выводить результаты в графической форме. Есть пара книг по «научному программированию на Python», но в них речь идет о численных расчетах и инженерных приложениях: решении дифференциальных уравнений, вычислении интегралов, моделировании методом Монте-Карло и различных вопросах, относящихся скорее к математике, чем к анализу данных и статистике. Поскольку в этой книге рассказывается о том, как научиться профессионально работать с данными в Python, я думаю, имеет смысл потратить некоторое время на объяснение наиболее важных возможностей встроенных структур данных и стандартных библиотек Python с точки зрения обработки структурированных и неструктурированных данных. Здесь представлено ровно столько информации, сколько необходимо для чтения книги.

Это приложение следует рассматривать не как исчерпывающее введение в язык Python, а как краткий – без излишеств – обзор тех средств, которые многократно используются в книге. Тем, кто только начинает писать на Python, я рекомендую дополнить его чтением официального руководства (*http://docs.python.org*) и, быть может, какой-нибудь из великолепных (и куда более объемных) книг по общим вопросам программирования на этом языке. На мой взгляд, совершенно необязательно быть профессиональным разработчиком качественного программного обеспечения на Python, чтобы продуктивно работать в области анализа данных. Призываю вас активнее использовать IPython для экспериментов с примерами кода и изучать различные типы, функции и методы по документации.

Значительная часть этой книги посвящена средствам высокопроизводительных вычислений на базе массивов при работе с большими наборами данных. Чтобы воспользоваться этими средствами, часто необходимо предварительно преобразовать замусоренные данные в структурированную форму, более пригодную для обработки. По счастью, Python позволяет быстро привести данные к надлежащему

виду, и в этом смысле он один из самых простых для работы языков. Чем свободнее вы владеете языком, тем меньше сложностей возникнет при подготовке новых наборов данных для анализа.

Интерпретатор Python

Python – *интерпретируемый* язык. Интерпретатор Python исполняет программу по одному предложению за раз. Стандартный интерактивный интерпретатор Python запускается из командной строки командой `python`:

```
$ python
Python 2.7.2 (default, Oct 4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 5
>>> print a
5
```

Строка `>>>` – это приглашение к вводу выражения. Для выхода из интерпретатора Python и возврата в командную строку нужно либо ввести команду `exit()`, либо нажать `Ctrl-D`.

Для выполнения Python-программы нужно просто набрать команду `python`, указав в качестве первого аргумента имя файла с расширением `.py`. Допустим, вы создали файл `hello_world.py` с таким содержимым:

```
print 'Hello world'
```

Чтобы выполнить его, достаточно ввести следующую команду:

```
$ python hello_world.py
Hello world
```

Многие программисты выполняют свой код на Python именно таким образом, но в мире *научных* приложений принято использовать IPython, улучшенный и дополненный интерпретатор Python. Ему посвящена вся глава 3. С помощью команды `%run` IPython исполняет код в указанном файле в том же процессе, что позволяет интерактивно изучать результаты по завершении выполнения.

```
$ ipython
Python 2.7.2 |EPD 7.1-2 (64-bit)| (default, Jul 3 2011, 15:17:51)
Type "copyright", "credits" or "license" for more information.

IPython 0.12 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %run hello_world.py
```

```
Hello world

In [2]:
```

По умолчанию приглашение IPython содержит не стандартную строку >>>, а строку вида `In [2]:`, включающую порядковый номер предложения.

Основы

Семантика языка

Структура языка Python отличается удобочитаемостью, простотой и ясностью. Некоторые даже называют написанный на Python код «исполняемым псевдокодом».

Отступы вместо скобок

В Python для структурирования кода используются пробелы (или знаки табуляции), а не фигурные скобки, как во многих других языках, например, R, C++, Java и Perl. Вот как выглядит цикл в алгоритме быстрой сортировки:

```
for x in array:
  if x < pivot:
     less.append(x)
  else:
     greater.append(x)
```

Двоеточием обозначается начало блока кода с отступом, весь последующий код до конца блока должен быть набран с точно таким же отступом. В каком-нибудь другом языке тот же код был бы записан так:

```
for x in array {
  if x < pivot {
     less.append(x)
  } else {
     greater.append(x)
  }
}
```

Основная причина такого использования пробелов заключается в стремлении сделать любой код на Python структурно единообразным, чтобы у программиста не возникал когнитивный диссонанс при чтении кода, написанного не им (или им, но давно и в спешке!). В языках без синтаксически значимых пробелов тот же код мог бы быть отформатирован иначе:

```
for x in array
{
  if x < pivot
  {
     less.append(x)
  }
```

```
    else
    {
       greater.append(x)
    }
}
```

Нравится вам это или нет, но синтаксически значимые пробелы – факт, с которым программисты на Python должны смириться, но по собственному опыту могу сказать, что благодаря им код на Python выглядит гораздо более удобочитаемым, чем на других знакомых мне языках. Поначалу такой стиль может показаться чужеродным, но со временем вы привыкнете и полюбите его.

Я настоятельно рекомендую использовать *4 пробела* в качестве величины отступа по умолчанию и настроить редактор так, чтобы он заменял знаки табуляции четырьмя пробелами. Некоторые используют знаки табуляции непосредственно или задают другое число пробелов, например, довольно часто встречаются отступы шириной 2. Но 4 пробела – соглашение, принятое подавляющим большинством программистов на Python, и я советую его придерживаться, если нет каких-то противопоказаний.

Вы уже поняли, что предложения в Python не обязаны завершаться точкой с запятой. Но ее можно использовать, чтобы отделить друг от друга предложения, находящиеся в одной строчке:

```
a = 5; b = 6; c = 7
```

Впрочем, писать несколько предложений в одной строчке не рекомендуется, потому что код из-за этого становится труднее читать.

Всё является объектом

Важная характеристика языка Python – последовательность его *объектной модели*. Все числа, строки, структуры данных, функции, классы, модули и т. д. в интерпретаторе заключены в «ящики», которые называются *объектами Python*. С каждым объектом ассоциирован *тип* (например, *строка* или *функция*) и внутренние данные. На практике это делает язык более гибким, потому что даже функции можно рассматривать как объекты.

Комментарии

Интерпретатор Python игнорирует текст, которому предшествует знак решетки `#`. Часто этим пользуются, чтобы включить в код комментарии. Иногда желательно исключить какие-то блоки кода, не удаляя их. Самое простое решение – *закомментировать* такой код:

```
results = []
for line in file_handle:
    # пока оставляем пустые строки
    # if len(line) == 0:
    #   continue
  results.append(line.replace('foo', 'bar'))
```

Вызов функции и метода объекта

После имени функции ставятся круглые скобки, внутри которых может быть нуль или более параметров. Возвращенное значение может быть присвоено переменной:

```
result = f(x, y, z)
g()
```

Почти со всеми объектами в Python ассоциированы функции, которые имеют доступ к внутреннему состоянию объекта и называются *методами*. Синтаксически вызов методов выглядит так:

```
obj.some_method(x, y, z)
```

Функции могут принимать *позиционные* и *именованные* аргументы:

```
result = f(a, b, c, d=5, e='foo')
```

Подробнее об этом ниже.

Переменные и передача по ссылке

Присваивание значения переменной (или *имени*) в Python приводит к созданию *ссылки* на объект, стоящий в правой части присваивания. Рассмотрим список целых чисел:

```
In [241]: a = [1, 2, 3]
```

Предположим, что мы присвоили значение `a` новой переменной `b`:

```
In [242]: b = a
```

В некоторых языках такое присваивание приводит к копированию данных `[1, 2, 3]`. В Python `a` и `b` указывают на один и тот же объект – исходный список `[1, 2, 3]` (это схематически изображено на рис. П.1). Чтобы убедиться в этом, добавим в список `a` еще один элемент и проверим затем список `b`:

```
In [243]: a.append(4)

In [244]: b
Out[244]: [1, 2, 3, 4]
```

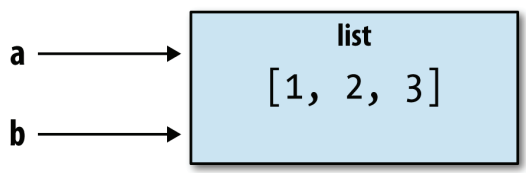


Рис. П.1. Две ссылки на один объект

Понимать семантику ссылок в Python и знать, когда, как и почему данные копируются, особенно важно при работе с большими наборами данных.

Операцию присваивания называют также *связыванием*, потому что мы связываем имя с объектом. Имена переменных, которым присвоено значение, иногда называют связанными переменными.

Передавая объекты в качестве аргументов функции, мы передаем только ссылки – копирование не производится. Поэтому говорят, что в Python аргументы передаются *по ссылке*, тогда как в некоторых других языках возможна передача как по ссылке, так и по значению (с созданием копий). Это означает, что функция может изменять значения своих аргументов. Пусть есть такая функция:

```
def append_element(some_list, element):
  some_list.append(element)
```

С учетом сказанного выше следующее поведение не должно вызывать удивления:

```
In [2]: data = [1, 2, 3]

In [3]: append_element(data, 4)

In [4]: data
Out[4]: [1, 2, 3, 4]
```

Динамические ссылки, строгие типы

В отличие от многих компилируемых языков, в частности Java и C++, со *ссылкой* на объект в Python не связан никакой тип. Следующий код не приведет к ошибке:

```
In [245]: a = 5              In [246]: type(a)
                             Out[246]: int

In [247]: a = 'foo'          In [248]: type(a)
                             Out[248]: str
```

Переменные – это имена объектов в некотором пространстве имен; информация о типе хранится в самом объекте. Отсюда некоторые делают поспешный вывод, будто Python не является «типизированным языком». Это не так, рассмотрим следующий пример:

```
In [249]: '5' + 5
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-249-f9dbf5f0b234> in <module>()
----> 1 '5' + 5
TypeError: cannot concatenate 'str' and 'int' objects
```

В некоторых языках, например в Visual Basic, строка `'5'` могла бы быть неявно преобразована (*приведена*) в целое число, и это выражение было бы вычислено как 10. А бывают и такие языки, например JavaScript, где целое число `5` преобразуется в строку, после чего производится конкатенация – `'55'`. В этом отношении Python считается *строго типизированным* языком, т. е. у любого объекта есть конкретный тип (или *класс*), а неявные преобразования разрешены только в некоторых не вызывающих сомнений случаев, например:

```
In [250]: a = 4.5

In [251]: b = 2

# Форматирование строки, см. ниже
In [252]: print 'a is %s, b is %s' % (type(a), type(b))
a is <type 'float'>, b is <type 'int'>

In [253]: a / b
Out[253]: 2.25
```

Знать тип объекта важно, и полезно также уметь писать функции, способные обрабатывать входные параметры различных типов. Проверить, что объект является экземпляром определенного типа, позволяет функция `isinstance`:

```
In [254]: a = 5        In [255]: isinstance(a, int)
                       Out[255]: True
```

Функция `isinstance` может также принимать кортеж типов и тогда проверяет, что тип переданного объекта присутствует в кортеже:

```
In [256]: a = 5; b = 4.5

In [257]: isinstance(a, (int, float))  In [258]: isinstance(b, (int, float))
Out[257]: True                          Out[258]: True
```

Атрибуты и методы

Объекты в Python обычно имеют атрибуты – другие объекты, хранящиеся «внутри» данного, – и методы – ассоциированные с объектом функции, имеющие доступ к внутреннему состоянию объекта. Обращение к тем и другим синтаксически выглядит как *obj.attribute_name*:

```
In [1]: a = 'foo'

In [2]: a.<Tab>
a.capitalize   a.format     a.isupper     a.rindex      a.strip
a.center       a.index      a.join        a.rjust       a.swapcase
a.count        a.isalnum    a.ljust       a.rpartition  a.title
a.decode       a.isalpha    a.lower       a.rsplit      a.translate
a.encode       a.isdigit    a.lstrip      a.rstrip      a.upper
a.endswith     a.islower    a.partition   a.split       a.zfill
a.expandtabs   a.isspace    a.replace     a.splitlines
a.find         a.istitle    a.rfind       a.startswith
```

К атрибутам и методам можно обращаться также с помощью функции `getattr`:

```
>>> getattr(a, 'split')
<function split>
```

Хотя в этой книге мы почти не использовали функцию `getattr`, а также родственные ей `hasattr` и `setattr`, они весьма эффективны для написания обобщенного, повторно используемого кода.

Динамическая типизация

Часто нас интересует не тип объекта, а лишь наличие у него определенных методов или поведения. Например, объект поддерживает итерирование, если он реализует *протокол итератора*. Для многих объектов это означает, что имеется «магический метод» `__iter__`, хотя есть другой – и лучший – способ проверки: попробовать воспользоваться функций `iter`:

```
def isiterable(obj):
  try:
     iter(obj)
     return True
  except TypeError: # not iterable
     return False
```

Эта функция возвращает `True` для строк, а также для большинства типов коллекций в Python:

```
In [260]: isiterable('a string')
Out[260]: True

In [261]: isiterable([1, 2, 3])
Out[261]: True

In [262]: isiterable(5)
Out[262]: False
```

Эту функциональность я постоянно использую для написания функций, принимающих параметры разных типов. Типичный случай – функция, принимающая любую последовательность (список, кортеж, ndarray) или даже итератор. Можно сначала проверить, является ли объект списком (или массивом NumPy), и, если нет, преобразовать его в таковой:

```
if not isinstance(x, list) and isiterable(x):
  x = list(x)
```

Импорт

В Python *модуль* – это просто файл с расширением `.py`, который содержит функции и различные определения, в том числе импортированные из других `py`-файлов. Пусть имеется следующий модуль:

```
# some_module.py
PI = 3.14159

def f(x):
```

```
    return x + 2

def g(a, b):
    return a + b
```

Если бы мы захотели обратиться к переменным или функциям, определенным в some_module.py, из другого файла в том же каталоге, то должны были бы написать:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Или эквивалентно:

```
from some_module import f, g, PI
result = g(5, PI)
```

Ключевое слово as позволяет переименовать импортированные сущности:

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Бинарные операторы и операции сравнения

Большинство математических операций и операций сравнения именно таковы, как мы и ожидаем:

```
In [263]: 5 - 7          In [264]: 12 + 21.5
Out[263]: -2             Out[264]: 33.5

In [265]: 5 <= 2
Out[265]: False
```

Список всех бинарных операторов приведен в табл. П.1.

Таблица П.1. Бинарные операторы

Операция	Описание
a + b	Сложить a и b
a - b	Вычесть b из a
a * b	Умножить a на b
a / b	Разделить a на b
a // b	Разделить a на b нацело, отбросив дробный остаток
a ** b	Возвести a в степень b
a & b	True, если и a, и b равны True. Для целых чисел вычисляет поразрядное И

Операция	Описание
`a | b`	`True`, либо `a`, либо `b` равно `True`. Для целых чисел вычисляет поразрядное ИЛИ
`a ^ b`	Для булевых величин `True`, если либо `a`, либо `b`, но не обе одновременно равны `True`. Для целых чисел вычисляет поразрядное ИСКЛЮЧИТЕЛЬНОЕ ИЛИ
`a == b`	`True`, если `a` равно `b`
`a != b`	`True`, если `a` не равно `b`
`a < b, a <= b`	`True`, если `a` меньше (меньше или равно) `b`
`a > b, a >= b`	`True`, если `a` больше (больше или равно) `b`
`a is b`	`True`, если `a` и `b` ссылаются на один и тот же объект Python
`a is not b`	`True`, если `a` и `b` ссылаются на разные объекты Python

Для проверки того, что две ссылки ведут на один и тот же объект, служит оператор `is`. Оператор `is not` тоже существует и позволяет проверить, что два объекта различаются.

```
In [266]: a = [1, 2, 3]

In [267]: b = a

# Обратите внимание: функция list всегда создает новый список
In [268]: c = list(a)

In [269]: a is b        In [270]: a is not c
Out[269]: True          Out[270]: True
```

Отметим, что это не то же самое, что сравнение с помощью оператора `==`, потому что в данном случае мы получим:

```
In [271]: a == c
Out[271]: True
```

Операторы `is` и `is not` очень часто употребляются, чтобы проверить, равна ли некоторая переменная `None`, потому что существует ровно один экземпляр `None`:

```
In [272]: a = None

In [273]: a is None
Out[273]: True
```

Немедленное и отложенное вычисление

В любом языке программирования важно понимать, *когда* вычисляется выражение. Рассмотрим простое выражение:

```
a = b = c = 5
d = a + b * c
```

В Python вычисление производится сразу после разбора выражения, и переменная `d` получает значение 30. При использовании другой парадигмы программирования, например в чистом функциональном языке, каким является Haskell, значение `d` может не вычисляться, пока не потребуется в каком-то другом месте. Это так называемое *отложенное*, или *ленивое* вычисление. Но Python в этом отношение очень *энергичный* язык. Почти всегда вычисления производятся немедленно. Даже в приведенном выше простом выражении произведение `b * c` вычисляет отдельным шагом до сложения с `a`.

В Python существует механизмы, в особенности итераторы и генераторы, позволяющие реализовать отложенные вычисления. Если вычисление обходится очень дорого, а необходимо не всегда, то такая техника может принести заметную пользу приложениям для обработки данных.

Изменяемые и неизменяемые объекты

Большинство объектов в Python – списки, словари, массивы NumPy и почти все определенные пользователем типы (классы) – изменяемы. Это означает, что объект или значение, которое в нем хранится, можно модифицировать.

```
In [274]: a_list = ['foo', 2, [4, 5]]

In [275]: a_list[2] = (3, 4)

In [276]: a_list
Out[276]: ['foo', 2, (3, 4)]
```

Но некоторые объекты, например строки и кортежи, неизменяемы:

```
In [277]: a_tuple = (3, 5, (4, 5))

In [278]: a_tuple[1] = 'four'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-278-b7966a9ae0f1> in <module>()
----> 1 a_tuple[1] = 'four'
TypeError: 'tuple' object does not support item assignment
```

Помните, что *возможность* изменять объект не означает *необходимости* это делать. Подобные действия в программировании называются *побочными эффектами*. Когда вы пишете функцию, обо всех ее побочных эффектах следует сообщать пользователю в комментариях или в документации. Я рекомендую по возможности избегать побочных эффектов, *отдавая предпочтение неизменяемости*, даже если вы работаете с изменяемыми объектами.

Скалярные типы

В Python есть небольшой набор встроенных типов для работы с числовыми данными, строками, булевыми значениями (`True` и `False`), датами и временем. Перечень основных скалярных типов приведен в табл. П.2. Работа с датами и вре-

менем будет рассмотрена отдельно, потому что эти типы определены в стандартном модуле `datetime`.

Таблица П.2. Стандартные скалярные типы в Python

Тип	Описание
`None`	Значение «null» в Python (существует только один экземпляр объекта `None`)
str	Тип строки. В Python 2.x может содержать только символы кодировки ASCII, в Python 3 – любые символы Unicode
`unicode`	Тип Unicode-строки
`float`	Число с плавающей точкой двойной точности (64-разрядное). Отдельный тип `double` не предусмотрен
bool	Значение `True` или `False`
`int`	Целое со знаком, максимальное значение зависит от платформы
`long`	Целое со знаком произвольной точности. Большие значения типа `int` автоматически преобразуются в `long`

Числовые типы

Основные числовые типы в Python – `int` и `float`. Размер целого числа, которое может быть представлено типом `int`, зависит от платформы (32- или 64-разрядной), но Python автоматически преобразует слишком большие целые в тип `long`, способный представить сколь угодно большое целое число.

```
In [279]: ival = 17239871

In [280]: ival ** 6
Out[280]: 26254519291092456596965462913230729701102721L
```

Числа с плавающей точкой представляются типом Python `float`, который реализован в виде значения двойной точности (64-разрядного). Такие числа можно записывать и в научной нотации:

```
In [281]: fval = 7.243

In [282]: fval2 = 6.78e-5
```

В версии Python 3 деление целых чисел, результатом которого не является целое число, дает число с плавающей точкой:

```
In [284]: 3 / 2
Out[284]: 1.5
```

В версии Python 2.7 и более ранних (с которыми, скорее всего, будут работать некоторые читатели) такое поведение можно обеспечить по умолчанию, поместив в начало своего модуля следующее загадочно выглядящее предложение:

```
from __future__ import division
```

А если вы не хотите это делать, то всегда можно явно преобразовать знаменатель в число с плавающей точкой:

```
In [285]: 3 / float(2)
Out[285]: 1.5
```

Для выполнения целочисленного деления в духе языка C (когда дробная часть результата отбрасывается) служит оператор деления с отбрасыванием `//`:

```
In [286]: 3 // 2
Out[286]: 1
```

Комплексные числа записывают с `j` в обозначении мнимой части:

```
In [287]: cval = 1 + 2j

In [288]: cval * (1 - 2j)
Out[288]: (5+0j)
```

Строки

Многие используют Python за его мощные и гибкие средства работы со строками. *Строковый литерал* записывается в одиночных (') или двойных (") кавычках:

```
a = 'one way of writing a string'
b = "another way"
```

Для записи многострочных строк, содержащих разрывы, используются тройные кавычки – `'''` или `"""`:

```
c = """
Это длинная строка,
занимающая несколько строчек
"""
```

Строки в Python неизменяемы, при любой модификации создается новая строка:

```
In [289]: a = 'this is a string'

In [290]: a[10] = 'f'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-290-5ca625d1e504> in <module>()
----> 1 a[10] = 'f'
TypeError: 'str' object does not support item assignment

In [291]: b = a.replace('string', 'longer string')

In [292]: b
Out[292]: 'this is a longer string'
```

Многие объекты Python можно преобразовать в строку с помощью функции `str`:

```
In [293]: a = 5.6 In [294]: s = str(a)

In [295]: s
Out[295]: '5.6'
```

Строки – это последовательности символов и потому могут рассматриваться как любые другие последовательности, например списки или кортежи:

```
In [296]: s = 'python'            In [297]: list(s)
                                  Out[297]: ['p', 'y', 't', 'h', 'o', 'n']

In [298]: s[:3]
Out[298]: 'pyt'
```

Знак обратной косой черты \ играет роль *управляющего символа*, он предшествует специальным символам, например знаку новой строки или символам Unicode. Чтобы записать строковый литерал, содержащий знак обратной косой черты, этот знак необходимо повторить дважды:

```
In [299]: s = '12\\34'

In [300]: print s
12\34
```

Если строка содержит много знаков обратной косой черты и ни одного специального символа, то при такой записи она становится совершенно неразборчивой. По счастью, есть другой способ: поставить перед начальной кавычкой букву r, которая означает, что все символы должны интерпретироваться буквально:

```
In [301]: s = r'this\has\no\special\characters'

In [302]: s
Out[302]: 'this\\has\\no\\special\\characters'
```

Сложение двух строк означает конкатенацию, при этом создается новая строка:

```
In [303]: a = 'this is the first half '

In [304]: b = 'and this is the second half'

In [305]: a + b
Out[305]: 'this is the first half and this is the second half'
```

Еще одна важная тема – форматирование строк. С появлением Python 3 диапазон возможностей в этом плане расширился, здесь я лишь вкратце опишу один из основных интерфейсов. Строка, содержащая знак %, за которым следует один или несколько символов формата, является шаблоном для подстановки значений (в точности, как в функции printf в C). Для примера рассмотрим такую строку:

```
In [306]: template = '%.2f %s are worth $%d'
```

Здесь %s означает, что аргумент нужно форматировать как строку, %.2f – как число с двумя десятичными знаками после запятой, а %d – как целое. Для подстановки аргументов вместо этих форматных параметров используется бинарный оператор %, за которым следует кортеж значений:

```
In [307]: template % (4.5560, 'Argentine Pesos', 1)
Out[307]: '4.56 Argentine Pesos are worth $1'
```

Форматирование строк – обширная тема; существует несколько методов и многочисленные параметры и ухищрения, призванные контролировать, как именно должны форматироваться значения, подставляемые в результирующую строку. Подробные сведения можно найти в Интернете. В главе 7 обсуждаются общие вопросы работы со строками в контексте анализа данных.

Булевы значения

Два булевых значения записываются в Python как True и False. Результатом сравнения и вычисления условных выражений является True или False. Булевы значения объединяются с помощью ключевых слов and и or:

```
In [308]: True and True
Out[308]: True

In [309]: False or True
Out[309]: True
```

Почти все встроенные объекты Python, а также объект любого класса, в котором определен магический метод __nonzero__, можно интерпретировать в логическом контексте, создаваемом предложением if:

```
In [310]: a = [1, 2, 3]
   .....: if a:
   .....:    print 'Я что-то нашел!'
   .....:
Я что-то нашел!

In [311]: b = []
   .....: if not b:
   .....:    print Пусто!'
   .....:
Пусто!
```

У большинства объектов в Python имеется понятие истинности или ложности. Например, пустая последовательность (список, словарь, кортеж и т. д.) трактуется как False, если встречается в логическом контексте (как в примере пустого списка b выше). Чтобы узнать, какое булево значение соответствует объекту, нужно передать его функции bool:

```
In [312]: bool([]), bool([1, 2, 3])
Out[312]: (False, True)

In [313]: bool('Hello world!'), bool('')
```

```
Out[313]: (True, False)

In [314]: bool(0), bool(1)
Out[314]: (False, True)
```

Приведение типов

Типы `str`, `bool`, `int` и `float` являются также функциями, которые можно использовать для приведения значения к соответствующему типу:

```
In [315]: s = '3.14159'

In [316]: fval = float(s)        In [317]: type(fval)
                                 Out[317]: float

In [318]: int(fval)        In [319]: bool(fval)       In [320]: bool(0)
Out[318]: 3                Out[319]: True             Out[320]: False
```

Тип None

`None` – это тип значения null в Python. Если функция явно не возвращает никакого значения, то неявно она возвращает `None`.

```
In [321]: a = None          In [322]: a is None
                            Out[322]: True

In [323]: b = 5             In [324]: b is not None
                            Out[324]: True
```

`None` также часто применяется в качестве значения по умолчанию для необязательных аргументов функции:

```
def add_and_maybe_multiply(a, b, c=None):
  result = a + b
  if c is not None:
     result = result * c
  return result
```

Хотя это вопрос чисто технический, стоит иметь в виду, что `None` – не зарезервированное слово языка, а единственный экземпляр класса `NoneType`.

Дата и время

Стандартный модуль Python `datetime` предоставляет типы `datetime`, `date` и `time`. Тип `datetime`, как нетрудно сообразить, объединяет информацию, хранящуюся в `date` и `time`. Именно он чаще всего и используется:

```
In [325]: from datetime import datetime, date, time

In [326]: dt = datetime(2011, 10, 29, 20, 30, 21)

In [327]: dt.day            In [328]: dt.minute
Out[327]: 29                Out[328]: 30
```

Имея экземпляр `datetime`, можно получить из него объекты `date` и `time` путем вызова одноименных методов:

```
In [329]: dt.date()                      In [330]: dt.time()
Out[329]: datetime.date(2011, 10, 29)    Out[330]: datetime.time(20, 30, 21)
```

Метод `strftime` форматирует объект `datetime`, представляя его в виде строки:

```
In [331]: dt.strftime('%m/%d/%Y %H:%M')
Out[331]: '10/29/2011 20:30'
```

Чтобы разобрать строку и представить ее в виде объекта `datetime`, нужно вызвать функцию `strptime`:

```
In [332]: datetime.strptime('20091031', '%Y%m%d')
Out[332]: datetime.datetime(2009, 10, 31, 0, 0)
```

В табл. 10.2 приведен полный перечень спецификаций формата.

При агрегировании или еще какой-то группировке временных рядов иногда бывает полезно заменить некоторые компоненты даты или времени, например, обнулить минуты и секунды, создав новый объект:

```
In [333]: dt.replace(minute=0, second=0)
Out[333]: datetime.datetime(2011, 10, 29, 20, 0)
```

Вычитание объектов `datetime` дает объект типа `datetime.timedelta`:

```
In [334]: dt2 = datetime(2011, 11, 15, 22, 30)

In [335]: delta = dt2 - dt

In [336]: delta                          In [337]: type(delta)
Out[336]: datetime.timedelta(17, 7179)   Out[337]: datetime.timedelta
```

Сложение объектов `timedelta` и `datetime` дает новый объект `datetime`, отстоящий от исходного на указанный промежуток времени:

```
In [338]: dt
Out[338]: datetime.datetime(2011, 10, 29, 20, 30, 21)

In [339]: dt + delta
Out[339]: datetime.datetime(2011, 11, 15, 22, 30)
```

Поток управления

if, elif, else

Предложение `if` – одно из самых хорошо известных предложений управления потоком выполнения. Оно вычисляет условие и, если получилось `True`, исполняет код в следующем далее блоке:

```
if x < 0:
  print 'Отрицательно'
```

После предложения `if` может находиться один или несколько блоков `elif` и блок `else`, который выполняется, если все остальные условия оказались равны `False`:

```
if x < 0:
  print 'Отрицательно'
elif x == 0:
  print 'Равно нулю'
elif 0 < x < 5:
  print 'Положительно, но меньше 5'
else:
  print 'Положительно и больше или равно 5'
```

Если какое-то условие равно `True`, последующие блоки `elif` и `else` даже не рассматриваются. В случае составного условия, в котором есть операторы `and` или `or`, частичные условия вычисляются слева направо с закорачиванием:

```
In [340]: a = 5; b = 7

In [341]: c = 8; d = 4

In [342]: if a < b or c > d:
   .....:    print 'Сделано'
Сделано
```

В этом примере условие `c > d` не вычисляется, потому что уже первое сравнение `a < b` равно `True`.

Циклы for

Циклы `for` предназначены для обхода коллекции (например, списка или кортежа) или итератора. Стандартный синтаксис выглядит так:

```
for value in collection:
  # что-то сделать с value
```

Ключевое слово `continue` позволяет сразу перейти к следующей итерации цикла, не доходя до конца блока. Рассмотрим следующий код, который суммирует целые числа из списка, пропуская значения `None`:

```
sequence = [1, 2, None, 4, None, 5]
total = 0
for value in sequence:
  if value is None:
     continue
  total += value
```

Из цикла `for` можно выйти досрочно с помощью ключевого слова `break`. Следующий код суммирует элементы списка, пока не встретится число 5:

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
```

```
    if value == 5:
        break
    total_until_5 += value
```

Если элементы коллекции или итераторы являются последовательностями (например, кортежем или списком), то их можно *распаковать* в переменные:

```
for a, b, c in iterator:
    # что-то сделать
```

Циклы while

Цикл `while` состоит из условия и блока кода, который выполняется до тех пор, пока условие не окажется равным `False` или не произойдет выход из цикла в результате предложения `break`:

```
x = 256
total = 0
while x > 0:
    if total > 500:
        break
    total += x
    x = x // 2
```

Ключевое слово pass

Предложение `pass` Python является «пустышкой». Его можно использовать в тех блоках, в которых не требуется никакое действие; нужно оно только потому, что в Python ограничителем блока выступает пробел:

```
if x < 0:
    print 'отрицательно!'
elif x == 0:
    # TODO: сделать тут что-нибудь разумное
    pass
else:
    print 'положительно!'
```

Часто предложение `pass` оставляют, имея в виду, что позже оно будет заменено какой-то полезной функциональностью.

```
def f(x, y, z):
    # TODO: реализовать эту функцию!
    pass
```

Обработка исключений

Обработка ошибок, или *исключений* в Python – важная часть создания надежных программ. В приложениях для анализа данных многие функции работают только для входных данных определенного вида. Например, функция `float` может привести строку к типу числа с плавающей точкой, но если формат строки заведомо некорректен, то завершается с ошибкой `ValueError`:

```
In [343]: float('1.2345')
Out[343]: 1.2345

In [344]: float('something')
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-344-439904410854> in <module>()
----> 1 float('something')
ValueError: could not convert string to float: something
```

Пусть требуется написать версию `float`, которая не завершается с ошибкой, а возвращает поданный на вход аргумент. Это можно сделать, обернув вызов `float` блоком `try/except`:

```
def attempt_float(x):
  try:
     return float(x)
  except:
     return x
```

Код в части `except` будет выполняться, только если `float(x)` возбуждает исключение:

```
In [346]: attempt_float('1.2345')
Out[346]: 1.2345

In [347]: attempt_float('something')
Out[347]: 'something'
```

Кстати, `float` может возбуждать и другие исключения, не только `ValueError`:

```
In [348]: float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-348-842079ebb635> in <module>()
----> 1 float((1, 2))
TypeError: float() argument must be a string or a number
```

Возможно, вы хотите перехватить только исключение `ValueError`, поскольку `TypeError` (аргумент был не строкой и не числом) может свидетельствовать о логической ошибке в программе. Для этого нужно написать после `except` тип исключения:

```
def attempt_float(x):
  try:
     return float(x)
  except ValueError:
     return x
```

В таком случае получим:

```
In [350]: attempt_float((1, 2))
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-350-9bdfd730cead> in <module>()
----> 1 attempt_float((1, 2))
<ipython-input-349-3e06b8379b6b> in attempt_float(x)
      1 def attempt_float(x):
      2     try:
----> 3         return float(x)
      4     except ValueError:
      5         return x
TypeError: float() argument must be a string or a number
```

Можно перехватывать исключения нескольких типов, для этого достаточно написать кортеж типов (скобки обязательны):

```
def attempt_float(x):
  try:
     return float(x)
  except (TypeError, ValueError):
     return x
```

Иногда исключение не нужно перехватывать, но какой-то код должен быть выполнен вне зависимости от того, возникло исключение в блоке `try` или нет. Для этого служит предложение `finally`:

```
f = open(path, 'w')
try:
  write_to_file(f)
finally:
  f.close()
```

Здесь описатель файла `f` закрывается *в любом случае*. Можно также написать код, который исполняется, только если в блоке `try` не было исключения, для этого используется ключевое слово `else`:

```
f = open(path, 'w')
try:
  write_to_file(f)
except:
  print 'Ошибка'
else:
  print 'Все хорошо'
finally:
  f.close()
```

Функции range и xrange

Функция `range` порождает список равноотстоящих целых чисел:

```
In [352]: range(10)
Out[352]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Можно задать начало, конец и шаг диапазона:

```
In [353]: range(0, 20, 2)
Out[353]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Как видите, конечная точка в порождаемый диапазон `range` не включается. Типичное применение `range` – обход последовательности по индексу:

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
  val = seq[i]
```

Если диапазон очень длинный, то рекомендуется использовать функцию `xrange`, которая принимает те же аргументы, что `range`, но возвращает итератор, который генерирует целые числа по одному, а не сразу все с запоминанием в гигантском списке. Следующий код вычисляет сумму тех чисел от 0 до 9999, которые кратны 3 или 5:

```
sum = 0
for i in xrange(10000):
  # % — оператор деления по модулю
  if x % 3 == 0 or x % 5 == 0:
     sum += i
```

В Python 3 функция `range` всегда возвращает итератор, поэтому использовать `xrange` необязательно.

Тернарные выражения

Тернарное выражение в Python позволяет записать блок `if-else`, порождающий единственное значение, в виде однострочного выражения. Синтаксически это выглядит так:

```
value = true-expr if condition else false-expr
```

Здесь *true-expr* и *false-expr* могут быть произвольными выражениями Python. Результат такой же, как при выполнении более пространной конструкции:

```
if condition:
  value = true-expr
else:
  value = false-expr
```

Вот более конкретный пример:

```
In [354]: x = 5

In [355]: 'Неотрицательно' if x >= 0 else 'Отрицательно'
Out[355]: 'Неотрицательно'
```

Как и в случае блоков `if-else`, вычисляется только одно из двух подвыражений. И хотя возникает искушение использовать тернарные выражения всегда, чтобы сократить длину программы, нужно понимать, что таким образом вы приносите в жертву понятность кода, если подвыражения достаточно сложны.

Структуры данных и последовательности

Структуры данных в Python просты, но эффективны. Чтобы стать хорошим программистом на Python, необходимо овладеть ими в совершенстве.

Кортеж

Кортеж – это одномерная *неизменяемая* последовательность объектов Python фиксированной длины. Проще всего создать кортеж, записав последовательность значений через запятую:

```
In [356]: tup = 4, 5, 6

In [357]: tup
Out[357]: (4, 5, 6)
```

Когда кортеж определяется в более сложном выражении, часто бывает необходимо заключать значения в скобки, как в следующем примере, где создается кортеж кортежей:

```
In [358]: nested_tup = (4, 5, 6), (7, 8)

In [359]: nested_tup
Out[359]: ((4, 5, 6), (7, 8))
```

Любую последовательность или итератор можно преобразовать в кортеж с помощью функции `tuple`:

```
In [360]: tuple([4, 0, 2])
Out[360]: (4, 0, 2)

In [361]: tup = tuple('string')

In [362]: tup
Out[362]: ('s', 't', 'r', 'i', 'n', 'g')
```

К элементам кортежа можно обращаться с помощью квадратных скобок `[]`, как и для большинства других типов последовательностей. Как и в C, C++, Java и многих других языках, нумерация элементов последовательностей в Python начинается с нуля:

```
In [363]: tup[0]
Out[363]: 's'
```

Хотя объекты, хранящиеся в кортеже, могут быть изменяемыми, сам кортеж после создания изменить (т. е. записать что-то другое в существующую позицию) невозможно:

```
In [364]: tup = tuple(['foo', [1, 2], True])

In [365]: tup[2] = False
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-365-c7308343b841> in <module>()
----> 1 tup[2] = False
TypeError: 'tuple' object does not support item assignment
# however
In [366]: tup[1].append(3)
In [367]: tup
Out[367]: ('foo', [1, 2, 3], True)
```

Кортежи можно конкатенировать с помощью оператора +, получая в результате более длинный кортеж:

```
In [368]: (4, None, 'foo') + (6, 0) + ('bar',)
Out[368]: (4, None, 'foo', 6, 0, 'bar')
```

Умножение кортежа на целое число, как и в случае списка, приводит к конкатенации нескольких копий кортежа.

```
In [369]: ('foo', 'bar') * 4
Out[369]: ('foo', 'bar', 'foo', 'bar', 'foo', 'bar', 'foo', 'bar')
```

Отметим, что копируются не сами объекты, а только ссылки на них.

Распаковка кортежей

При попытке *присвоить* значение похожему на кортеж выражению, состоящему из нескольких переменных, интерпретатор пытается *распаковать* значение в правой части оператора присваивания:

```
In [370]: tup = (4, 5, 6)

In [371]: a, b, c = tup

In [372]: b
Out[372]: 5
```

Распаковать можно даже вложенные кортежи:

```
In [373]: tup = 4, 5, (6, 7)

In [374]: a, b, (c, d) = tup

In [375]: d
Out[375]: 7
```

Эта функциональность позволяет без труда решить задачу обмена значений переменных, которая во многих других языках решается так:

```
tmp = a
a = b
b = tmp
```

тогда как на Python достаточно написать:

```
b, a = a, b
```

Одно из наиболее распространенных применений распаковки переменных – обход последовательности кортежей или списков:

```
seq = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
for a, b, c in seq:
  pass
```

Другое применение – возврат нескольких значений из функции. Подробнее об этом ниже.

Методы кортежа

Поскольку ни размер, ни содержимое кортежа нельзя модифицировать, методов экземпляра у него совсем немного. Пожалуй, наиболее полезен метод `count` (имеется также у списков), возвращающий количество вхождений значения:

```
In [376]: a = (1, 2, 2, 2, 3, 4, 2)

In [377]: a.count(2)
Out[377]: 4
```

Список

В отличие от кортежей, списки имеют переменную длину, а их содержимое можно модифицировать. Список определяется с помощью квадратных скобок `[]` или конструктора типа `list`:

```
In [378]: a_list = [2, 3, 7, None]

In [379]: tup = ('foo', 'bar', 'baz')

In [380]: b_list = list(tup)
                                  In [381]: b_list
                                  Out[381]: ['foo', 'bar', 'baz']

In [382]: b_list[1] = 'peekaboo'
                                  In [383]: b_list
                                  Out[383]: ['foo', 'peekaboo', 'baz']
```

Семантически списки и кортежи схожи, поскольку те и другие являются одномерными последовательностями объектов. Поэтому во многих функциях они взаимозаменяемы.

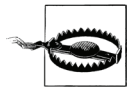
Добавление и удаление элементов

Для добавления элемента в конец списка служит метод `append`:

```
In [384]: b_list.append('dwarf')

In [385]: b_list
Out[385]: ['foo', 'peekaboo', 'baz', 'dwarf']
```

Метод `insert` позволяет вставить элемент в указанную позицию списка:

```
In [386]: b_list.insert(1, 'red')

In [387]: b_list
Out[387]: ['foo', 'red', 'peekaboo', 'baz', 'dwarf']
```



Метод `insert` вычислительно сложнее, чем `append`, так как чтобы освободить место для нового элемента, приходится сдвигать ссылки на элементы, следующие за ним.

Операцией, обратной к `insert`, является `pop`, она удаляет из списка элемент, находившийся в указанной позиции, и возвращает его:

```
In [388]: b_list.pop(2)
Out[388]: 'peekaboo'

In [389]: b_list
Out[389]: ['foo', 'red', 'baz', 'dwarf']
```

Элементы можно удалять также по значению методом `remove`, который находит и удаляет из списка первый элемент с указанным значением:

```
In [390]: b_list.append('foo')

In [391]: b_list.remove('foo')

In [392]: b_list
Out[392]: ['red', 'baz', 'dwarf', 'foo']
```

Если снижение производительности из-за использования методов `append` и `remove`, не составляет проблемы, то список Python вполне можно использовать в качестве структуры данных «мультимножество». Чтобы проверить, содержит ли список некоторое значение, используется ключевое слово `in`:

```
In [393]: 'dwarf' in b_list
Out[393]: True
```

Отметим, что проверка вхождения значения в случае списка занимает гораздо больше времени, чем в случае словаря или множества, потому что Python должен просматривать список от начала до конца, а это требует линейного времени, тогда как поиск в других структурах занимает постоянное время.

Конкатенация и комбинирование списков

Как и в случае кортежей, операция сложения конкатенирует списки:

```
In [394]: [4, None, 'foo'] + [7, 8, (2, 3)]
Out[394]: [4, None, 'foo', 7, 8, (2, 3)]
```

Если уже имеется список, то добавить в его конец несколько элементов позволяет метод `extend`:

```
In [395]: x = [4, None, 'foo']

In [396]: x.extend([7, 8, (2, 3)])

In [397]: x
Out[397]: [4, None, 'foo', 7, 8, (2, 3)]
```

Отметим, что конкатенация – сравнительно дорогая операция, потому что нужно создать новый список и скопировать в него все объекты. Обычно предпочтительнее использовать `extend` для добавления элементов в существующий список, особенно если строится длинный список. Таким образом

```
everything = []
for chunk in list_of_lists:
  everything.extend(chunk)
```

быстрее чем эквивалентная конкатенация:

```
everything = []
for chunk in list_of_lists:
  everything = everything + chunk
```

Сортировка

Список можно отсортировать на месте (без создания нового объекта), вызвав его метод `sort`:

```
In [398]: a = [7, 2, 5, 1, 3]

In [399]: a.sort()

In [400]: a
Out[400]: [1, 2, 3, 5, 7]
```

У метода `sort` есть несколько удобных возможностей. Одна из них – возможность передать *ключ сортировки*, т. е. функцию, порождающую значение, по которому должны сортироваться объекты. Например, вот как можно отсортировать коллекцию строк по длине:

```
In [401]: b = ['saw', 'small', 'He', 'foxes', 'six']

In [402]: b.sort(key=len)

In [403]: b
Out[403]: ['He', 'saw', 'six', 'small', 'foxes']
```

Двоичный поиск и поддержание списка в отсортированном состоянии

В стандартном модуле `bisect` реализованы операции двоичного поиска и вставки в отсортированный список. Метод `bisect.bisect` находит позицию, в

которую следует вставить новый элемент, чтобы список остался отсортированным, а метод `bisect.insort` производит вставку в эту позицию:

```
In [404]: import bisect

In [405]: c = [1, 2, 2, 2, 3, 4, 7]

In [406]: bisect.bisect(c, 2) In [407]: bisect.bisect(c, 5)
Out[406]: 4 Out[407]: 6

In [408]: bisect.insort(c, 6)

In [409]: c
Out[409]: [1, 2, 2, 2, 3, 4, 6, 7]
```

Функции из модуля `bisect` не проверяют, отсортирован ли исходный список, т. к. это обошлось бы дорого с вычислительной точки зрения. Поэтому их применение к неотсортированному списку завершается без ошибок, но результат может оказаться неверным.

Вырезание

Из спископодобных коллекций (массивов NumPy, кортежей) можно вырезать участки с помощью нотации, которая в простейшей форме сводится к передаче пары `start:stop` оператору доступа по индексу `[]`:

```
In [410]: seq = [7, 2, 3, 7, 5, 6, 0, 1]

In [411]: seq[1:5]
Out[411]: [2, 3, 7, 5]
```

Срезу также можно присваивать последовательность:

```
In [412]: seq[3:4] = [6, 3]

In [413]: seq
Out[413]: [7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Элемент с индексом `start` включается в срез, элемент с индексом `stop` не включается, поэтому количество элементов в результате равно `stop - start`.

Любой член пары – как `start`, так и `stop` – можно опустить, тогда по умолчанию подразумевается начало и конец последовательности соответственно:

```
In [414]: seq[:5] In [415]: seq[3:]
Out[414]: [7, 2, 3, 6, 3] Out[415]: [6, 3, 5, 6, 0, 1]
```

Если индекс в срезе отрицателен, то он отсчитывается от конца последовательности:

```
In [416]: seq[-4:]          In [417]: seq[-6:-2]
Out[416]: [5, 6, 0, 1]      Out[417]: [6, 3, 5, 6]
```

К семантике вырезания надо привыкнуть, особенно если вы раньше работали с R или MATLAB. На рис. П.2 показано, как происходит вырезание при положительном и отрицательном индексе.

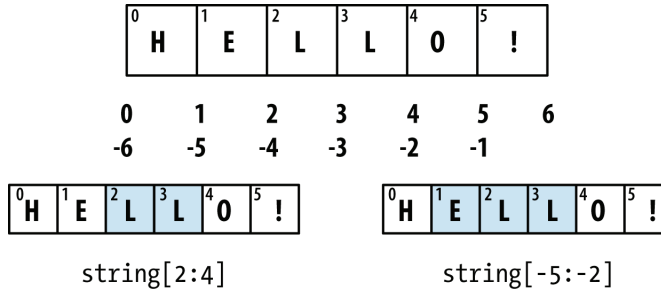


Рис. П.2. Иллюстрация соглашений о вырезании в Python

Допускается и вторая запятая, после которой можно указать шаг, например, взять каждый второй элемент:

```
In [418]: seq[::2]
Out[418]: [7, 3, 3, 6, 1]
```

Если задать шаг `-1`, то список или кортеж будет инвертирован:

```
In [419]: seq[::-1]
Out[419]: [1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Встроенные функции последовательностей

У последовательностей в Python есть несколько полезных функций, которые следует знать и применять при любой возможности.

enumerate

При обходе последовательности часто бывает необходимо следить за индексом текущего элемента. «Ручками» это можно сделать так:

```
i = 0
for value in collection:
  # что-то сделать с value
  i += 1
```

Но поскольку эта задача встречается очень часто, в Python имеется встроенная функция `enumerate`, которая возвращает последовательность кортежей `(i, value)`:

```
for i, value in enumerate(collection):
  # что-то сделать с value
```

Функция `enumerate` нередко используется для построения словаря, отображающего значения в последовательности (предполагаемые уникальными) на их позиции:

```
In [420]: some_list = ['foo', 'bar', 'baz']

In [421]: mapping = dict((v, i) for i, v in enumerate(some_list))

In [422]: mapping
Out[422]: {'bar': 1, 'baz': 2, 'foo': 0}
```

sorted

Функция `sorted` возвращает новый отсортированный список, построенный из элементов произвольной последовательности:

```
In [423]: sorted([7, 1, 2, 6, 0, 3, 2])
Out[423]: [0, 1, 2, 2, 3, 6, 7]

In [424]: sorted('horse race')
Out[424]: [' ', 'a', 'c', 'e', 'e', 'h', 'o', 'r', 'r', 's']
```

Для получения отсортированного списка уникальных элементов последовательности нередко используют `sorted` совместно с `set`:

```
In [425]: sorted(set('this is just some string'))
Out[425]: [' ', 'e', 'g', 'h', 'i', 'j', 'm', 'n', 'o', 'r', 's', 't', 'u']
```

zip

Функция `zip` «сшивает» элементы нескольких список, кортежей или других последовательностей в пары, создавая список кортежей:

```
In [426]: seq1 = ['foo', 'bar', 'baz']

In [427]: seq2 = ['one', 'two', 'three']

In [428]: zip(seq1, seq2)
Out[428]: [('foo', 'one'), ('bar', 'two'), ('baz', 'three')]
```

Функция `zip` принимает любое число аргументов, а количество порождаемых ей кортежей определяется длиной *самой короткой* последовательности:

```
In [429]: seq3 = [False, True]

In [430]: zip(seq1, seq2, seq3)
Out[430]: [('foo', 'one', False), ('bar', 'two', True)]
```

Очень распространенное применение `zip` – одновременный обход нескольких последовательностей, возможно, в сочетании с `enumerate`:

```
In [431]: for i, (a, b) in enumerate(zip(seq1, seq2)):
    .....:     print('%d: %s, %s' % (i, a, b))
    .....:
0: foo, one
1: bar, two
2: baz, three
```

Если имеется «сшитая» последовательность, то `zip` можно использовать, чтобы «распороть» ее. Это можно также представлять себе как преобразование списка *строк* в список *столбцов*. Синтаксис, несколько причудливый, выглядит следующим образом:

```
In [432]: pitchers = [('Nolan', 'Ryan'), ('Roger', 'Clemens'),
   .....:             ('Schilling', 'Curt')]

In [433]: first_names, last_names = zip(*pitchers)

In [434]: first_names
Out[434]: ('Nolan', 'Roger', 'Schilling')

In [435]: last_names
Out[435]: ('Ryan', 'Clemens', 'Curt')
```

Более подробно мы поговорим об использовании оператора `*`, когда будем рассматривать вызовы функций. Эта запись эквивалентна следующей:

```
zip(seq[0], seq[1], ..., seq[len(seq) - 1])
```

reversed

Функция `reversed` перебирает элементы последовательности в обратном порядке:

```
In [436]: list(reversed(range(10)))
Out[436]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Словарь

Словарь, пожалуй, является самой важной из встроенных в Python структур данных. Его также называют *хэшем*, *отображением* или *ассоциативным массивом*. Он представляет собой коллекцию пар *ключ–значение* переменного размера, в которой и ключ, и значение – объекты Python. Создать словарь можно с помощью фигурных скобок `{}`, отделяя ключи от значений двоеточием:

```
In [437]: empty_dict = {}

In [438]: d1 = {'a' : 'some value', 'b' : [1, 2, 3, 4]}

In [439]: d1
Out[439]: {'a': 'some value', 'b': [1, 2, 3, 4]}
```

Для доступа к элементам, вставки и присваивания применяется такой же синтаксис, как в случае списка или кортежа:

```
In [440]: d1[7] = 'an integer'

In [441]: d1
Out[441]: {7: 'an integer', 'a': 'some value', 'b': [1, 2, 3, 4]}
```

```
In [442]: d1['b']
Out[442]: [1, 2, 3, 4]
```

Проверка наличия ключа в словаре тоже производится, как для кортежа или списка:

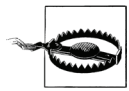
```
In [443]: 'b' in d1
Out[443]: True
```

Для удаления ключа можно использовать либо ключевое слово `del`, либо метод `pop` (который не только удаляет ключ, но и возвращает ассоциированное с ним значение):

```
In [444]: d1[5] = 'some value'

In [445]: d1['dummy'] = 'another value'

In [446]: del d1[5]

In [447]: ret = d1.pop('dummy')          In [448]: ret
                                         Out[448]: 'another value'
```

Методы `keys` и `values` возвращают соответственно список ключей и список значений. Хотя точный порядок пар ключ–значение не определен, эти методы возвращают ключи и значения в одном и том же порядке:

```
In [449]: d1.keys()      In [450]: d1.values()
Out[449]: ['a', 'b', 7] Out[450]: ['some value', [1, 2, 3, 4], 'an integer']
```



В версии Python 3 `dict.keys()` и `dict.values()` – не списки, а итераторы.

Два словаря можно объединить в один методом `update`:

```
In [451]: d1.update({'b' : 'foo', 'c' : 12})

In [452]: d1
Out[452]: {7: 'an integer', 'a': 'some value', 'b': 'foo', 'c': 12}
```

Создание словаря из последовательностей

Нередко бывает, что имеются две последовательности, которые естественно рассматривать как ключи и соответствующие им значения, а, значит, требуется построить из них словарь. Первая попытка могла бы выглядеть так:

```
mapping = {}
for key, value in zip(key_list, value_list):
  mapping[key] = value
```

Поскольку словарь – это, по существу, коллекция 2-кортежей, не удивительно, что функция типа `dict` принимает список 2-кортежей:

```
In [453]: mapping = dict(zip(range(5), reversed(range(5))))

In [454]: mapping
Out[454]: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
```

В следующем разделе мы рассмотрим *словарное включение* – еще один элегантный способ построения словарей.

Значения по умолчанию

Очень часто можно встретить код, реализующий такой логику:

```
if key in some_dict:
  value = some_dict[key]
else:
  value = default_value
```

Поэтому методы словаря `get` и `pop` могут принимать значение, возвращаемое по умолчанию, так что этот блок `if-else` можно упростить:

```
value = some_dict.get(key, default_value)
```

Метод `get` по умолчанию возвращает `None`, если ключ не найден, тогда как `pop` в этом случае возбуждает исключение. Часто бывает, что значениями в словаре являются другие коллекции, например списки. Так, можно классифицировать слова по первой букве и представить их набор в виде словаря списков:

```
In [455]: words = ['apple', 'bat', 'bar', 'atom', 'book']

In [456]: by_letter = {}

In [457]: for word in words:
   .....:     letter = word[0]
   .....:     if letter not in by_letter:
   .....:         by_letter[letter] = [word]
   .....:     else:
   .....:         by_letter[letter].append(word)
   .....:
In [458]: by_letter
Out[458]: {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

Метод `setdefault` предназначен специально для этой цели. Блок `if-else` выше можно переписать так:

```
by_letter.setdefault(letter, []).append(word)
```

В стандартном модуле `collections` есть полезный класс `defaultdict`, который еще больше упрощает решение этой задачи. Его конструктору передается тип или функция, генерирующая значение по умолчанию для каждой пары в словаре:

```
from collections import defaultdict
by_letter = defaultdict(list)
for word in words:
  by_letter[word[0]].append(word)
```

Единственное, что требуется от инициализатора `defaultdict`, – быть объектом, допускающим вызов (например, функцией), причем необязательно конструктором типа. Следовательно, чтобы значение по умолчанию было равно 4, мы можем передать функцию, возвращающую 4:

```
counts = defaultdict(lambda: 4)
```

Допустимые типы ключей словаря

Значениями словаря могут быть произвольные объекты Python, но ключами должны быть неизменяемые объекты, например скалярные типы (int, float, строка) или кортежи (причем все объекты кортежа тоже должны быть неизменяемыми). Технически это свойство называется *хэшируемостью*. Проверить, является ли объект хэшируемым (и, стало быть, может быть ключом словаря), позволяет функция `hash`:

```
In [459]: hash('string')
Out[459]: -9167918882415130555

In [460]: hash((1, 2, (2, 3)))
Out[460]: 1097636502276347782

In [461]: hash((1, 2, [2, 3])) # ошибка, списки изменяемы
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-461-800cd14ba8be> in <module>()
----> 1 hash((1, 2, [2, 3]))    # ошибка, списки изменяемы
TypeError: unhashable type: 'list'
```

Чтобы использовать список в качестве ключа, достаточно преобразовать его в кортеж:

```
In [462]: d = {}

In [463]: d[tuple([1, 2, 3])] = 5

In [464]: d
Out[464]: {(1, 2, 3): 5}
```

Множество

Множество – это неупорядоченная коллекция уникальных элементов. Можно считать, что это словари, не содержащие значений. Создать множество можно двумя способами: с помощью функции `set` или задав *множество-литерал* в фигурных скобках:

```
In [465]: set([2, 2, 2, 1, 3, 3])
Out[465]: set([1, 2, 3])

In [466]: {2, 2, 2, 1, 3, 3}
Out[466]: set([1, 2, 3])
```

Множества поддерживают теоретико-множественные операции: объединение, пересечение, разность и симметрическая разность. Наиболее употребительные методы множеств перечислены в табл. П.3.

```
In [467]: a = {1, 2, 3, 4, 5}

In [468]: b = {3, 4, 5, 6, 7, 8}

In [469]: a | b # объединение (or)
Out[469]: set([1, 2, 3, 4, 5, 6, 7, 8])

In [470]: a & b # пересечение (and)
Out[470]: set([3, 4, 5])

In [471]: a - b # разность
Out[471]: set([1, 2])

In [472]: a ^ b # симметрическая разность (xor)
Out[472]: set([1, 2, 6, 7, 8])
```

Таблица П.3. Операции над множествами в Python

Функция	Альтернативный синтаксис	Описание
`a.add(x)`	Нет	Добавить элемент `x` в множество `a`
`a.remove(x)`	Нет	Удалить элемент `x` из множества `a`
`a.union(b)`	`a | b`	Найти все уникальные элементы, входящие либо в `a`, либо в `b`
`a.intersection(b)`	`a & b`	Найти все элементы, входящие и в `a`, и в `b`
`a.difference(b)`	`a - b`	Найти элементы, входящие в `a`, но не входящие в `b`
`a.symmetric_difference(b)`	`a ^ b`	Найти элементы, входящие либо в `a`, либо в `b`, но не в `a` и `b` одновременно
`a.issubset(b)`	Нет	`True`, если все элементы `a` входят также и в `b`
`a.issuperset(b)`	Нет	`True`, если все элементы `b` входят также и в `a`
`a.isdisjoint(b)`	Нет	`True`, если у `a` и `b` нет ни одного общего элемента

Можно также проверить, является ли множество подмножеством (содержится в) или надмножеством (содержит) другого множества:

```
In [473]: a_set = {1, 2, 3, 4, 5}

In [474]: {1, 2, 3}.issubset(a_set)
Out[474]: True

In [475]: a_set.issuperset({1, 2, 3})
Out[475]: True
```

Как нетрудно догадаться, множества называются равными, если состоят из одинаковых элементов:

```
In [476]: {1, 2, 3} == {3, 2, 1}
Out[476]: True
```

Списковое, словарное и множественное включение

Списковое включение[1] – одна из самых любимых особенностей Python. Этот механизм позволяет кратко записать создание нового списка, образованного фильтрацией элементов коллекции с одновременным преобразованием элементов, прошедших через фильтр. Основная синтаксическая форма такова:

```
[expr for val in collection if condition]
```

Это эквивалентно следующему циклу `for`:

```
result = []
for val in collection:
  if condition:
    result.append(expr)
```

Условие фильтрации можно опустить, оставив только выражение. Например, если задан список строк, то мы могли бы выделить из него строки длиной больше 2 и попутно преобразовать их в верхний регистр:

```
In [477]: strings = ['a', 'as', 'bat', 'car', 'dove', 'python']

In [478]: [x.upper() for x in strings if len(x) > 2]
Out[478]: ['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Словарное и множественное включение – естественные обобщения, которые предлагают аналогичную идиому для порождения словарей и множеств. Словарное включение выглядит так:

```
dict_comp = {key-expr : value-expr for value in collection if condition}
```

Множественное включение очень похоже на списковое, то квадратные скобки заменяются фигурными:

```
set_comp = {expr for value in collection if condition}
```

Все виды включений – не более чем синтаксическая глазурь, упрощающая чтение и написание кода. Рассмотрим приведенный выше список строк. Допустим, что требуется построить множество, содержащее длины входящих в коллекцию строк; это легко сделать с помощью множественного включения:

[1] Более-менее устоявшийся перевод термина list comprehension — «списковое включение» — крайне неудачен и совершенно не отражает суть дела. Я предложил бы перевод «трансфильтрация», являющееся объединением слов «трансформация» и «фильтрация», но не уверен в реакции сообщества. – *Прим. перев.*

```
In [479]: unique_lengths = {len(x) for x in strings}

In [480]: unique_lengths
Out[480]: set([1, 2, 3, 4, 6])
```

В качестве простого примера словарного включения создадим словарь, сопоставляющий каждой строке ее позицию в списке:

```
In [481]: loc_mapping = {val : index for index, val in enumerate(strings)}

In [482]: loc_mapping
Out[482]: {'a': 0, 'as': 1, 'bat': 2, 'car': 3, 'dove': 4, 'python': 5}
```

Этот словарь можно было бы построить и так:

```
loc_mapping = dict((val, idx) for idx, val in enumerate(strings)}
```

На мой взгляд, вариант со словарным включением короче и чище.

Словарное и множественное включение были добавлены в Python сравнительно недавно: в версиях Python 2.7 и Python 3.1+.

Вложенное списковое включение

Пусть имеется список списков, содержащий имена мальчиков и девочек:

```
In [483]: all_data = [['Tom', 'Billy', 'Jefferson', 'Andrew', 'Wesley',
  .....:               'Steven',     'Joe'],
  .....:              ['Susie', 'Casey', 'Jill', 'Ana', 'Eva', 'Jennifer',
  .....:               'Stephanie']]
```

Возможно, эти имена взяты из разных файлов, потому что вам нужно хранить мужские и женские имена по отдельности. А теперь допустим, что требуется получить один список, содержащий все имена, в которых встречается не менее двух букв `e`. Конечно, это можно было бы сделать в таком простом цикле `for`:

```
names_of_interest = []
for names in all_data:
  enough_es = [name for name in names if name.count('e') > 2]
  names_of_interest.extend(enough_es)
```

Но можно обернуть всю операцию одним *вложенным списковым включением*:

```
In [484]: result = [name for names in all_data for name in names
   .....:           if name.count('e') >= 2]

In [485]: result
Out[485]: ['Jefferson', 'Wesley', 'Steven', 'Jennifer', 'Stephanie']
```

Поначалу вложенное списковое включение с трудом укладывается в мозгу. Части `for` соответствуют порядку вложенности, а все фильтры располагаются в

конце, как и раньше. Вот еще один пример, в котором мы линеаризуем список кортежей целых чисел, создавая один плоский список:

```
In [486]: some_tuples = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]

In [487]: flattened = [x for tup in some_tuples for x in tup]

In [488]: flattened
Out[488]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Помните, что порядок выражений `for` точно такой же, как если бы вы писали вложенные циклы `for`, а не списковое включение:

```
flattened = []
for tup in some_tuples:
  for x in tup:
     flattened.append(x)
```

Глубина уровня вложенности не ограничена, хотя если уровней больше трех, стоит задуматься о правильности выбранной структуры данных. Важно отличать показанный выше синтаксис от спискового включения внутри спискового включения – тоже вполне допустимой конструкции:

```
In [229]: [[x for x in tup] for tup in some_tuples]
```

Функции

Функции – главный и самый важный способ организации и повторного использования кода в Python. Функций не может быть слишком много. На самом деле, я считаю, что большинство программистов, занимающихся анализом данных, пишут недостаточно функций! Как вы, конечно, поняли из примеров выше, объявление функции начинается ключевым словом `def`, а результат возвращается в предложении `return`:

```
def my_function(x, y, z=1.5):
  if z > 1:
     return z * (x + y)
  else:
     return z / (x + y)
```

Ничто не мешает иметь в функции несколько предложений `return`. Если при выполнении достигнут конец функции, а предложение `return` не встретилось, то возвращается `None`.

У функции могут быть *позиционные* и *именованные* аргументы. Именованные аргументы обычно используются для задания значений по умолчанию и необязательных аргументов. В примере выше `x` и `y` – позиционные аргументы, а `z` – именованный. Следующие вызовы функции эквивалентны:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
```

Основное ограничение состоит в том, именованные аргументы должны находиться после всех позиционных (если таковые имеются). Сами же именованные аргументы можно задавать в любом порядке, это освобождает программиста от необходимости помнить, в каком порядке были указаны аргументы функции в объявлении, важно лишь, как они называются.

Пространства имен, области видимости и локальные функции

Функции могут обращаться к переменным, объявленных в двух областях видимости: *глобальной* и *локальной*. Область видимости переменной в Python называют также *пространством имен*. Любая переменная, которой присвоено значение внутри функции, по умолчанию попадает в локальное пространство имен. Локальное пространство имен создается при вызове функции, и в него сразу же заносятся аргументы функции. По завершении функции локальное пространство имен уничтожается (хотя бывают и исключения, см. ниже раздел о замыканиях). Рассмотрим следующую функцию:

```
def func():
  a = []
  for i in range(5):
     a.append(i)
```

При вызове `func()` создается пустой список `a`, в него добавляется 5 элементов, а затем, когда функция завершается, список `a` уничтожается. Но допустим, что мы объявили `a` следующим образом:

```
a = []
def func():
  for i in range(5):
     a.append(i)
```

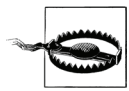
Присваивать значение глобальной переменной внутри функции допустимо, но такие переменные должны быть объявлены глобальными с помощью ключевого слова `global`:

```
In [489]: a = None

In [490]: def bind_a_variable():
.....:         global a
.....:         a = []
.....:     bind_a_variable()
.....:

In [491]: print a
[]
```

Функции можно объявлять в любом месте, в том числе допустимы *локальные* функции, которые динамически создаются внутри другой функции при ее вызове:

```
def outer_function(x, y, z):
  def inner_function(a, b, c):
     pass
  pass
```

Здесь функция `inner_function` не существует, пока не вызвана функция `outer_function`. Как только `outer_function` завершает выполнение, `inner_function` уничтожается.



Вообще говоря, я не рекомендую злоупотреблять ключевым словом `global`. Обычно глобальные переменные служат для хранения состояния системы. Если вы понимаете, что пользуетесь ими слишком часто, то стоит подумать о переходе к объектно-ориентированному программированию (использовать классы).

Вложенные функции могут обращаться к локальному пространству имен объемлющей функции, но не могут связывать в нем переменные. Подробнее об этом я расскажу в разделе о замыканиях.

Строго говоря, любая функция локальна в какой-то области видимости, хотя это может быть и область видимости на уровне модуля.

Возврат нескольких значений

Когда я только начинал программировать на Python после многих лет работы на Java и C++, одной из моих любимых была возможность возвращать из функции несколько значений. Вот простой пример:

```
def f():
  a = 5
  b = 6
  c = 7
  return a, b, c

a, b, c = f()
```

В анализе данных и других научных приложениях это встречается сплошь и рядом, потому что многие функции вычисляют несколько результатов. Вспомнив об упаковке и распаковке кортежей, о которых мы говорили выше в этом приложении, вы поймете, что именно это здесь и происходит: на самом деле, функция возвращает только *один* объект, кортеж, который затем распаковывается в результирующие переменные. Этот пример можно было бы записать и так:

```
return_value = f()
```

В таком случае `return_value` было бы 3-кортежем, содержащим все три возвращенные переменные. Иногда разумнее возвращать несколько значений не в виде кортежа, а в виде словаря:

```
def f():
  a = 5
```

```
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

Функции являются объектами

Поскольку функции в Python – объекты, становятся возможны многие конструкции, которые в других языках выразить трудно. Пусть, например, мы производим очистку данных и должны применить ряд преобразований к следующему списку строк:

```
states = [' Alabama ', 'Georgia!', 'Georgia', 'georgia', 'FlOrIda',
          'south carolina##', 'West virginia?']
```

Всякий, кому доводилось работать с присланными пользователями данными опроса, ожидает такого рода мусора. Чтобы сделать такой список строк пригодным для анализа, нужно произвести различные операции: удалить лишние пробелы и знаки препинания, оставить заглавные буквы только в нужных местах. Первая попытка могла бы выглядеть так:

```
import re    # Модуль регулярных выражений

def clean_strings(strings):
  result = []
  for value in strings:
     value = value.strip()
     value = re.sub('[!#?]', '', value) # удалить знаки препинания
     value = value.title()
     result.append(value)
  return result
```

Вот как выглядит результат:

```
In [15]: clean_strings(states)
Out[15]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Другой подход, который иногда бывает полезен, – составить список операций, которые необходимо применить к набору строк:

```
def remove_punctuation(value):
  return re.sub('[!#?]', '', value)

clean_ops = [str.strip, remove_punctuation, str.title]

def clean_strings(strings, ops):
  result = []
```

```
    for value in strings:
        for function in ops:
            value = function(value)
        result.append(value)
    return result
```

Далее поступаем следующим образом:

```
In [22]: clean_strings(states, clean_ops)
Out[22]:
['Alabama',
 'Georgia',
 'Georgia',
 'Georgia',
 'Florida',
 'South Carolina',
 'West Virginia']
```

Подобный *функциональный* подход позволяет задать способ модификации строк на очень высоком уровне. Степень повторной используемости функции `clean_strings` определенно возросла!

Функции можно передавать в качестве аргументов другим функциям, например встроенной функции `map`, которая применяет переданную функцию к коллекции:

```
In [23]: map(remove_punctuation, states)
Out[23]:
[' Alabama ',
 'Georgia',
 'Georgia',
 'georgia',
 'FlOrIda',
 'south carolina',
 'West virginia']
```

Анонимные (лямбда) функции

Python поддерживает так называемые *анонимные*, или *лямбда*-функции. По существу, это простые однострочные функции, возвращающие значение. Определяются они с помощью ключевого слова `lambda`, которое означает всего лишь «мы определяем анонимную функцию» и ничего более.

```
def short_function(x):
  return x * 2

equiv_anon = lambda x: x * 2
```

В этой книге я обычно употребляю термин «лямбда-функция». Они особенно удобны в ходе анализа данных, потому что, как вы увидите, во многих случаях функции преобразования данных принимают другие функции в качестве аргументов. Часто быстрее (и чище) передать лямбда-функцию, чем писать полноценное объявление функции или даже присваивать лямбда-функцию локальной переменной. Рассмотрим такой простенький пример:

```
def apply_to_list(some_list, f):
  return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Можно было бы, конечно, написать `[x * 2 for x in ints]`, но в данном случае нам удалось передать функции `apply_to_list` пользовательский оператор.

Еще пример: пусть требуется отсортировать коллекцию строк по количеству различных букв в строке.

```
In [492]: strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Для этого можно передать лямбда-функцию методу списка `sort`:

```
In [493]: strings.sort(key=lambda x: len(set(list(x))))

In [494]: strings
Out[494]: ['aaaa', 'foo', 'abab', 'bar', 'card']
```

Лямбда-функции называются анонимными, потому что объекту-функции не присваивается никакое имя.

Замыкания: функции, возвращающие функции

Бояться замыканий не надо. При определенных обстоятельствах это очень полезное и мощное средство! В двух словах, замыканием называется любая *динамически сгенерированная* функция, возвращенная из другой функции. Основное свойство замыкания состоит в том, что оно имеет доступ к переменным, определенным в том локальном пространстве имен, в котором было создано. Вот очень простой пример:

```
def make_closure(a):
  def closure():
    print('Я знаю секрет: %d' % a)
  return closure

closure = make_closure(5)
```

Разница между замыканием и обычной функцией Python состоит в том, что замыкание сохраняет доступ к пространству имен (функции), в котором было создано, даже если создавшая ее функция уже завершилась. Так, в примере выше, возвращенное замыкание печатает строку `Я знаю секрет: 5`, в какой бы момент ее ни вызвать. Хотя обычно создают замыкания со статическим внутренним состоянием (в данном случае только значение `a`), ничто не мешает включить в состав состояния изменяемый объект – словарь, множество, список – и затем модифицировать его. Например, ниже приведена функция, которая возвращает функцию, запоминающую, с какими аргументами вызывалась объемлющая функция:

```
def make_watcher():
  have_seen = {}

  def has_been_seen(x):
     if x in have_seen:
        return True
     else:
        have_seen[x] = True
     return False

  return has_been_seen
```

Вызвав эту функцию с несколькими целочисленными аргументами, получим:

```
In [496]: watcher = make_watcher()

In [497]: vals = [5, 6, 1, 5, 1, 6, 3, 5]

In [498]: [watcher(x) for x in vals]
Out[498]: [False, False, False, True, True, True, False, True]
```

Однако следует иметь в виду одно техническое ограничение: изменять внутреннее состояние объектов (например, добавлять в словарь пары ключ-значение) можно, но *связывать* переменные в области видимости объемлющей функции – нельзя. Обойти это ограничение можно, например, путем модификации словаря или списка вместо присваивания значений переменным.

```
def make_counter():
  count = [0]

  def counter():
     # увеличить счетчик и вернуть новое значение
     count[0] += 1
     return count[0]
  return counter

counter = make_counter()
```

Возникает вопрос, зачем все это нужно? На практике можно написать очень общую функцию с кучей параметров, а затем изготовить на ее основе более простые специализированные функции. Вот пример функции форматирования строк:

```
def format_and_pad(template, space):
  def formatter(x):
     return (template % x).rjust(space)

  return formatter
```

Затем можно создать функцию форматирования чисел с плавающей точкой, которая всегда возвращает строку длиной 15 символов:

```
In [500]: fmt = format_and_pad('%.4f', 15)

In [501]: fmt(1.756)
Out[501]: '         1.7560'
```

Углубившись в изучение объектно-ориентированного программирования на Python, вы поймете, что такой механизм можно реализовать (хотя и не столь лаконично) также с помощью классов.

Расширенный синтаксис вызова с помощью *args и **kwargs

Внутренний механизм обработки аргументов функции в Python весьма прост. Когда вы пишете `func(a, b, c, d=some, e=value)`, позиционные и именованные аргументы упаковываются в кортеж и словарь соответственно. Поэтому функция на самом деле получает кортеж `args` и словарь `kwargs` и делает примерно следующее:

```
a, b, c = args
d = kwargs.get('d', d_default_value)
e = kwargs.get('e', e_default_value)
```

Все это происходит за кулисами. Разумеется, функция проверяет корректность и позволяет задавать некоторые позиционные аргументы как именованные (даже если в объявлении функции для них нет имени!).

```
def say_hello_then_call_f(f, *args, **kwargs):
  print 'args is', args
  print 'kwargs is', kwargs
  print("Привет! Я сейчас вызову %s" % f)
  return f(*args, **kwargs)

def g(x, y, z=1):
  return (x + y) / z
```

Если мы теперь вызовем функцию `g` через `say_hello_then_call_f`, то получим:

```
In [8]: say_hello_then_call_f(g, 1, 2, z=5.)
args is (1, 2)
kwargs is {'z': 5.0}
Hello! Now I'm going to call <function g at 0x2dd5cf8>
Out[8]: 0.6
```

Каррирование: частичное фиксирование аргументов

В информатике термином *каррирование*[2] обозначается процедура порождения новых функций из существующих путем *фиксирования некоторых аргументов*. Пусть, например, имеется тривиальная функция сложения двух чисел:

```
def add_numbers(x, y):
  return x + y
```

[2] В честь американского математика и логика Хаскелла Брукса Карри. – *Прим. перев.*

Мы можем породить на ее основе новую функцию одной переменной, `add_five`, которая прибавляет к своему аргументу 5:

```
add_five = lambda y: add_numbers(5, y)
```

Говорят, что второй аргумент функции `add_numbers` *каррирован*. Ничего особо примечательного здесь нет, поскольку мы просто определили новую функцию, которая вызывает существующую. Стандартный модуль `functools` упрощает эту процедуру за счет функции `partial`:

```
from functools import partial
add_five = partial(add_numbers, 5)
```

При обсуждении библиотеки pandas мы будем пользоваться этой техникой для создания специализированных функций преобразования временных рядов:

```
# вычислить скользящее среднее временного ряда x за 60 дней
ma60 = lambda x: pandas.rolling_mean(x, 60)
# вычислить скользящие средние за 60 дней всех временных рядов в data
data.apply(ma60)
```

Генераторы

Наличие единого способа обхода последовательностей, например объектов в списке или строк в файле, – важная особенность Python. Реализована она с помощью протокола *итератора*, общего механизма, наделяющего объекты свойством итерируемости. Например, при обходе (итерировании) словаря мы получаем хранящиеся в нем ключи:

```
In [502]: some_dict = {'a': 1, 'b': 2, 'c': 3}

In [503]: for key in some_dict:
   .....:     print key,
a c b
```

Встречая конструкцию `for key in some_dict`, интерпретатор Python сначала пытается создать итератор из `some_dict`:

```
In [504]: dict_iterator = iter(some_dict)

In [505]: dict_iterator
Out[505]: <dictionary-keyiterator at 0x10a0a1578>
```

Итератор – это любой объект, который отдает интерпретатору Python объекты при использовании в контексте, аналогичном циклу `for`. Методы, ожидающие получить список или похожий на список объект, как правило, удовлетворяются любым итерируемым объектом. Это относится, в частности, к встроенным методам, например `min`, `max` и `sum`, и к конструкторам типов, например `list` и `tuple`:

```
In [506]: list(dict_iterator)
Out[506]: ['a', 'c', 'b']
```

Генератор – это простой способ конструирования итерируемого объекта. Если обычная функция выполняется и возвращает единственное значение, то генератор «лениво» возвращает последовательность значений, приостанавливаясь после возврата каждого в ожидании запроса следующего. Чтобы создать генератор, нужно вместо `return` использовать ключевое слово `yield`:

```
def squares(n=10):
  for i in xrange(1, n + 1):
     print 'Генерируются квадраты чисел от 1 до %d' % (n ** 2)
     yield i ** 2
```

В момент вызова генератора никакой код не выполняется:

```
In [2]: gen = squares()

In [3]: gen
Out[3]: <generator object squares at 0x34c8280>
```

И лишь после запроса элементов генератор начинает выполнять свой код:

```
In [4]: for x in gen:
   ...:     print x,
   ...:
Генерируются квадраты чисел от 1 до  100
1 4 9 16 25 36 49 64 81 100
```

В качестве не столь тривиального примера предположим, что требуется найти все способы разменять 1 доллар (100 центов) мелочью. Вы наверняка сможете придумать разные способы решения этой задачи и хранения уже встретившихся уникальных комбинаций. Один из них – написать генератор, который отдает списки монет (представленных целыми числами):

```
def make_change(amount, coins=[1, 5, 10, 25], hand=None):
  hand = [] if hand is None else hand
  if amount == 0:
     yield hand
  for coin in coins:
     # проверяем, что не отдаем слишком много мелочи и что все
     # все комбинации уникальны
     if coin > amount or (len(hand) > 0 and hand[-1] < coin):
        continue

     for result in make_change(amount - coin, coins=coins,
                               hand=hand + [coin]):
        yield result
```

Детали этого алгоритма не так существенны (сможете ли вы придумать более короткий?). Затем можно написать:

```
In [508]: for way in make_change(100, coins=[10, 25, 50]):
   .....:     print way
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
[25, 25, 10, 10, 10, 10, 10]
```

```
[25, 25, 25, 25]
[50, 10, 10, 10, 10, 10]
[50, 25, 25]
[50, 50]

In [509]: len(list(make_change(100)))
Out[509]: 242
```

Генераторные выражения

Создать генератор проще всего с помощью *генераторного выражения*. Такой генератор аналогичен списковому, словарному и множественному включению; чтобы его создать, заключите выражение, которое выглядит, как списковое включение, в круглые скобки вместо квадратных:

```
In [510]: gen = (x ** 2 for x in xrange(100))

In [511]: gen
Out[511]: <generator object <genexpr> at 0x10a0a31e0>
```

Это в точности эквивалентно следующему более многословному генератору:

```
def _make_gen():
  for x in xrange(100):
     yield x ** 2
  gen = _make_gen()
```

Генераторные выражения можно использовать внутри любой функции Python, принимающей генератор:

```
In [512]: sum(x ** 2 for x in xrange(100))
Out[512]: 328350

In [513]: dict((i, i **2) for i in xrange(5))
Out[513]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Модуль itertools

Стандартный библиотечный модуль `itertools` содержит набор генераторов для многих общеупотребительных алгоритмов. Так, генератор `groupby` принимает произвольную последовательность и функцию, он группирует соседние элементы последовательности по значению, возвращенному функцией, например:

```
In [514]: import itertools

In [515]: first_letter = lambda x: x[0]

In [516]: names = ['Alan', 'Adam', 'Wes', 'Will', 'Albert', 'Steven']

In [517]: for letter, names in itertools.groupby(names, first_letter):
   .....:     print letter, list(names)  # names - это генератор
A ['Alan', 'Adam']
W ['Wes', 'Will']
```

```
A ['Albert']
S ['Steven']
```

В табл. П.4 описаны некоторые функции из модуля itertools, которыми я часто пользуюсь.

Таблица П.4. Некоторые полезные функции из модуля itertools

Функция	Описание
`imap(func, *iterables)`	Генераторная версия встроенной функции `map`; применяет функцию `func` к каждому кортежу, «сшитому» из соответственных элементов переданных последовательностей
`ifilter(func, iterable)`	Генераторная версия встроенного фильтра; отдает элементы `x`, для которых `func(x)` равно `True`
`combinations(iterable, k)`	Генерирует последовательность всех возможных `k`-кортежей, составленных из элементов `iterable`, без учета порядка
`permutations(iterable, k)`	Генерирует последовательность всех возможных `k`-кортежей, составленных из элементов `iterable`, с учетом порядка
`groupby(iterable[, keyfunc])`	Генерирует пары (ключ, субитератор) для каждого уникального ключа

В Python 3 несколько встроенных функций (`zip`, `map`, `filter`), порождающих списки, заменены генераторными версиями, которые в Python 2 находятся в модуле `itertools`.

Файлы и операционная система

В этой книге для чтения файла с диска и загрузки данных из него в структуры Python как правило используются такие высокоуровневые средства, как функция `pandas.read_csv`. Однако важно понимать основы работы с файлами в Python. По счастью, здесь все очень просто, и именно поэтому Python так часто выбирают, когда нужно работать с текстом или файлами.

Чтобы открыть файл для чтения или для записи, пользуйтесь встроенной функцией `open`, которая принимает относительный или абсолютный путь:

```
In [518]: path = 'ch13/segismundo.txt'
In [519]: f = open(path)
```

По умолчанию файл открывается только для чтения – в режиме `'r'`. Далее описатель файла `f` можно рассматривать как список и перебирать строки:

```
for line in f:
  pass
```

У строк, прочитанных из файла, сохраняется признак конца строки (EOL), поэтому часто можно встретить код, который удаляет концы строк:

```
In [520]: lines = [x.rstrip() for x in open(path)]

In [521]: lines
Out[521]:
['Sue\xc3\xb1a el rico en su riqueza,',
 'que m\xc3\xa1s cuidados le ofrece;',
 '',
 'sue\xc3\xb1a el pobre que padece',
 'su miseria y su pobreza;',
 '',
 'sue\xc3\xb1a el que a medrar empieza,',
 'sue\xc3\xb1a el que afana y pretende,',
 'sue\xc3\xb1a el que agravia y ofende,',
 '',
 'y en el mundo, en conclusi\xc3\xb3n,',
 'todos sue\xc3\xb1an lo que son,',
 'aunque ninguno lo entiende.',
 '']
```

Если бы мы ввели команду `f = open(path, 'w')`, то был бы создан *новый файл* ch13/segismundo.txt, а старый с таким же именем был бы стерт. Все допустимые режимы ввода-вывода перечислены в табл. П.5.

Таблица П.5. Режимы открытия файла в Python

Режим	Описание
`r`	Режим чтения
`w`	Режим записи. Создается новый файл (а старый с тем же именем удаляется)
`a`	Дописывание в конец существующего файла (если файл не существует, он создается)
`r+`	Чтение и запись
`b`	Уточнение режима для двоичных файлов: `'rb'` или `'wb'`
`U`	Универсальный режим новых строк. Задается сам по себе (`'U'`) или как уточнение режима чтения (`'rU'`)

Для записи текста в файл служат методы `write` или `writelines`. Например, можно было бы создать вариант файла `prof_mod.py` без пустых строк:

```
In [522]: with open('tmp.txt', 'w') as handle:
   .....:     handle.writelines(x for x in open(path) if len(x) > 1)

In [523]: open('tmp.txt').readlines()
Out[523]:
['Sue\xc3\xb1a el rico en su riqueza,\n',
 'que m\xc3\xa1s cuidados le ofrece;\n',
 'sue\xc3\xb1a el pobre que padece\n',
 'su miseria y su pobreza;\n',
```

```
'sue\xc3\xb1a el que a medrar empieza,\n',
'sue\xc3\xb1a el que afana y pretende,\n',
'sue\xc3\xb1a el que agravia y ofende,\n',
'y en el mundo, en conclusi\xc3\xb3n,\n',
'todos sue\xc3\xb1an lo que son,\n',
'aunque ninguno lo entiende.\n']
```

В табл. П.6 приведены многие из наиболее употребительных методов работы с файлами.

Таблица П.6. Наиболее употребительные методы и атрибуты для работы с файлами в Python

Метод	Описание
`read([size])`	Возвращает прочитанные из файла данные в виде строки. Необязательный аргумент `size` говорит, сколько байтов читать
`readlines([size])`	Возвращает список прочитанных из файла строк. Необязательный аргумент `size` говорит, сколько строк читать
`write(str)`	Записывает переданную строку в файл
`writelines(strings)`	Записывает переданную последовательность строк в файл
`close()`	Закрывает описатель файла
`flush()`	Сбрасывает внутренний буфер ввода-вывода на диск
`seek(pos)`	Перемещает указатель чтения-записи на байт файла с указанным номером
`tell()`	Возвращает текущую позицию в файле в виде целого числа
`closed`	`True`, если файл закрыт

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

!

!cmd, команда 73

'

'heapsort', алгоритм сортировки 407

#

#, знак решетки 418

$

$PATH, переменная 23

%

% символ 428
%alias, магическая функция 74
%automagic, магическая функция 67
%a формат даты 320
%A формат даты 320
%bookmark, магическая функция 73, 75
%b формат даты 320
%B формат даты 320
%cd, магическая функция 73
%cpaste, магическая функция 64
%C формат даты и времени 320
%debug, магическая функция 66, 75
%dhist, магическая функция 73
%dirs, магическая функция 73
%d, спецификатор формата 429
%env, магическая функция 73
%gui, магическая функция 70
%hist, магическая функция 67, 72
%logstart, магическая функция 72
%logstop, магическая функция 72
%lprun, магическая функция 84, 85
%magic, магическая функция 67
%page, магическая функция 68
%paste, магическая функция 63, 67
%pdb, магическая функция 66, 76
%popd, магическая функция 73
%prun, магическая функция 68, 83
%pushd, магическая функция 73
%pwd, магическая функция 73
%p формат времени 320
%quickref, магическая функция 67
%reset, магическая функция 68, 72
%run, магическая функция 61, 68, 416
%s, спецификатор формата 429
%timeit, магическая функция 68, 80
%time, магическая функция 68, 80
%who_ls, магическая функция 68
%whos, магическая функция 68
%who, магическая функция 68
%xdel, магическая функция 68, 72
%xmode, магическая функция 66
%X формат времени 320
%x формат даты 320

(

(обратная косая черта) 428

.

.bash_profile, файл 21
.bashrc, файл 23

[

[] (квадратные скобки) 437, 439

_

_ (два знака подчеркивания) 71
_ (знак подчеркивания) 60, 71

{

{} (фигурные скобки) 445

>

>>> приглашение 416

А

агрегирование 115
агрегирование данных 285
  возврат данных в неиндексированном виде 289
  применение нескольких функций 287
аннотирование в matplotlib 254
анонимные функции 456
арифметические операции 146
  восполнение значений 147
  между DataFrame и Series 148
атрибуты
  в Python 421
  начинающиеся знаком подчеркивания 60

Б

база данных федеральной избирательной комиссии за 2012 год 305
  распределение суммы пожертвований по интервалам 311
  статистика пожертвований по роду занятий и месту работы 308
  статистика пожертвований по штатам 313
базы данных
  чтение и запись 196
бета-коэффициент 372
бета-распределение 123
библиотеки 16
  IPython 17
  matplotlib 17
  NumPy 16
  pandas 16
  SciPy 18
бинарные универсальные функции 111
биномиальное распределение 123
булевы
  индексирование массивов 104
  массивы 116
  тип данных 98, 429
буфер обмена, исполнение кода 63

В

векторизация 100
  определение 112
векторные строковые функции 235
вложенные типы данных 402
восполнение отсутствующих данных 164, 296
временные интервалы 316
временные метки
  использование периодов в качестве 363
  определение 316
  преобразование в периоды 339
временные ряды
  диапазоны дат 325
  и производительность 356
  класс TimeSeries 321
    выборка 322
    индексирование 322
    неуникальные индексы 324
  передискретизация. *См.* передискретизация
  периоды. *См.* периоды
  построение графиков 348
  сдвиг 329
  скользящие оконные функции. *См.* скользящие оконные функции
  типы данных 317
  часовые пояса 331
  частоты 326
    неделя месяца 329
выбросы, фильтрация 224
выравнивание данных 146, 358
  восполнение значений в арифметических методах 147
  операции между DataFrame и Series 148
вырезание
  в массивах 100
  в списках 442
выходные переменные 71

Г

гамма-распределение 123
генераторы 460
  генераторные выражения 462
  модуль itertools 462
генерация случайных чисел 122
гистограммы 264
глобальная блокировка интерпретатора (GIL) 15
глобальная область видимости 453
графики
  в pandas. *См.* pandas, построение графиков

временные ряды 348
построение с помощью matplotlib. *См.* matplotlib
пример набора данных о землетрясении на Гаити 267
графики плотности 264
графики ядерной оценки плотности (KDE) 265
группировка
group by, метод. *См.* group by, метод
агрегирование данных 285
применение нескольких функций 287
база данных федеральной избирательной комиссии за 2012 год 305
распределение суммы пожертвований по интервалам 311
статистика пожертвований по роду занятий и месту работы 308
статистика пожертвований по штатам 313
возврат данных в неиндексированном виде 289
восполнение отсутствующих данных 296
в финансовых приложениях 370
групповое взвешенное среднее 299
квантильный анализ 294
кросс-табуляция 302
линейная регрессия 301
метод apply 292
сводные таблицы 302
случайная выборка 297
групповые ключи 294

Д

дата и время
date_parser, аргумент 181
date_range, функция 325
DatetimeIndex, объект 138
dateutil, пакет 318
диапазоны дат 325
типы данных 317, 430
двоеточие 417
двоичные форматы данных 192
HDF5 193
Microsoft Excel 194
хранение массивов 119
двоичный поиск в списке 441
динамическая типизация в Python 422
динамически сгенерированные функции 457
дискретизация 222
доходность
индекс 368
кумулятивная 368
определение 368

З

завершение по нажатию клавиши Tab 59
закладки на каталоги в IPython 75
закрытые атрибуты 60
закрытые методы 60
замыкания 457
запись
в базу данных 196
в текстовый файл 182
землетрясение на Гаити, пример набора данных 267

И

иерархическое индексирование
в pandas 166
сводная статистика по уровню 170
столбцы DataFrame 170
уровни сортировки 169
изменение формы 213
изменение формы
массива 385
определение 212
с помощью иерархического индексирования 213
изменяемые объекты 425
именованные аргументы 419, 452
индексы
в pandas 154
в классе TimeSeries 321
массивов 100
определение 128
осей 221
слияние данных 204
целочисленный 172
индикаторные переменные 227
интегрированные среды разработки (IDE) 24, 64
исключения
автоматический вход в отладчик 67
обработка в Python 433
история команд, поиск 64
итератора протокол 422, 460

К

каррирование 459
квантильный анализ 294
квартильный анализ 373
ковариация 158
команды. *См. также* магические команды
  история в IPython 70
    входные и выходные переменные 71
    повторное выполнение 70
    протоколирование 72
  отладчика 78
  поиск 65
комбинации клавиш IPython 65
комбинации клавиш в IPython 64
комбинирование
  источников данных 366
  перекрывающихся данных 211
  списков 440
комментарии в Python 418
конкатенация
  вдоль оси 207
  массивов 388
контейнер однородных данных 401
конференции 24
концы интервалов 343
координированное универсальное время (UTC) 331
корреляция 158
кортежи 437
  методы 439
  распаковка 438
косвенная сортировка 405
кумулятивная доходность 368

Л

лексикографическая сортировка
  lexsort, метод 405
  определение 406
линеаризация 386
линейная алгебра 121
линейная регрессия 301, 380
линейные графики 258
локализация временных рядов 332
локальная область видимости 453
лямбда-функции 235, 288, 456

М

магические команды 66
магические методы 60
манипулирование данными
  изменение формы 213
  манипуляции со строками 229
    векторные строковые функции 235
    методы 230
    регулярные выражения 232
  поворот 215
  преобразование данных 217
    дискретизация 222
    замена значений 220
    индикаторные переменные 227
    переименование индексов осей 221
    перестановка 226
    с помощью функции или отображения 218
    устранение дубликатов 217
    фильтрация выбросов 224
  пример базы данных о продуктах питания 237
  слияние данных 200
    комбинирование перекрывающихся данных 211
    конкатенация вдоль оси 207
    слияние объектов DataFrame 200
    слияние по индексу 204
маргиналы 302
маркеры 250
массивы
  where, функция 113
  булево индексирование 104
  булевы 116
  в NumPy 385
    c_, объект 389
    r_, объект 389
    изменение формы 385
    конкатенация 388
    получение и установка подмножеств 391
    разбиение 388
    размещение в памяти 387
    репликация в памяти 390
    сохранение в файле 410
  вырезание 100
  индексы 100
  логические условия как операции с массивами 113
  операции между 100

перестановка осей 108
поиск в отсортированном массиве 407
прихотливое индексирование 107
создание 95
создание PeriodIndex 340
сортировка 117
статистические операции 115
структурные 401
вложенные типы данных 402
достоинства 403
манипуляции 403
типы данных 97
установка элементов с помощью укладывания 397
устранение дубликатов 118
файловый ввод-вывод 119
сохранение и загрузка текстовых файлов 120
хранение массивов на диске в двоичном формате 119
модули 422
момент первого пересечения 124

Н

надпериод 347
непрерывная память 412
непрерывный индекс доходности 378
нормализованный набор временных меток 326
нормальное распределение 123, 126

О

область видимости 453
обратная трассировка 65
объектная модель 418
Олсона база данных 331
оси
конкатенация вдоль 207
метки 252
переименование индексов 221
перестановка 108
укладывание по 394
отладчик IPython 75
отступы
IndentationError, исключение 63
в Python 417
отсутствующие данные 162
восполнение 164
фильтрация 163
очистка экрана, комбинация клавиш 65

П

панели 358
патчи 255
передача по ссылке 420
передискретизация 341
OHLC 344
методом groupby 345
периодов 346
повышающая 345
переменные среды 20, 73
перестановки 226
переформатирование 26
переформатирование данных 358
выборка по состоянию на 364
для нестандартных частот 361
комбинирование данных 366
с целью выравнивания 358
периоды 335
в качестве временных меток 363
квартальные 337
определение 316, 335
передискретизация 346
преобразование временных меток в 339
преобразование частоты 336
создание PeriodIndex из массивов 340
подграфики 246
подпериод 347
позиционные аргументы 419
пользовательские универсальные функции 400
понижающая передискретизация 341
посторонний столбец 280
поток управления 431
range, функция 435
xrange, функция 435
обработка исключений 433
предложение if 431
предложение pass 433
тернарное выражение 436
циклы for 432
циклы while 433
представления 101, 134
преобразование
между временными метками и периоды 339
между строкой и datetime 318

преобразование данных 217
  дискретизация 222
  замена значений 220
  индикаторные переменные 227
  переименование индексов осей 221
  перестановка 226
  с помощью функции или отображения 218
  устранение дубликатов 217
  фильтрация выбросов 224
прерывание программы 62, 65
приведение типа 98
приведение типов 430
прихотливое индексирование 107, 391
проецирование файла на память 410
промежуток 353
пространства имен 453
профили в IPython 90
псевдокод 26
пустое пространство имен 62

P

рабочий каталог 73
разбиение массивов 388
разделение-применение-объединение 277
ранжирование данных 151
регулярные выражения 232
редукция 155
репликация массивов 390
роллинг 378
роллинг фьючерсных контрактов 377

C

сводные статистики 155
  isin, метод 160
  unique, метод 160
  value_counts, метод 160
  корреляция и ковариация 158
  по уровню 170
сводные таблицы
  pivot, метод 215
  поворот данных 212
  таблицы сопряженности 304
связывание
  определение 420, 458
сдвиг временного ряда 329
системные команды, задание псевдонимов 73
скользящая корреляция 380
скользящие оконные функции 350
слияние данных 199
  комбинирование перекрывающихся 211
  конкатенация вдоль оси 207
  по индексу 204
  слияние объектов DataFrame 200
словари 445
  возврат переменных среды 73
  группировка с помощью 282
  значения по умолчанию 447
  ключи 448
  словарное включение 450
  создание 446
случайное блуждание 123
смещения во временных рядах 330
согласование с индексом 139
сортировка
  в NumPy 403
  в pandas 151
  массивов 117
  поиск в отсортированном массиве 407
  списков 441
  уровни 169
сортировка на месте 403
сортировки алгоритмы 406
списки 439
списковое включение 450
  вложенное 451
среднее с расширяющимся окном 351
срез 358
ссылки 419
статистические операции 115
стилизация в matplotlib 249
столбчатые диаграммы 260
стохастический граничный анализ 375
строго типизированные языки 421
строки
  преобразование в datetime 318
  тип данных 99, 427
    манипуляции 229
структурные массивы 401
  вложенные типы данных 402
  достоинства 403
  манипуляции 403
структуры данных в pandas 128
  DataFrame 131
  Index 137
  Panel 173
  Series 128

Т

текстовые редакторы, интеграция с IPython 64
текстовые файлы 175
  HTML-файлы 188
  lxml, библиотека 188
  XML-файлы 190
  вывод данных 182
  данные в формате JSON 186
  сохранение и загрузка 120
  формат с разделителями 184
  чтение порциями 181
тернарное выражение 436
типы данных
  в NumPy 384
  в Python 425
    None 430
    булев 429
    дата и время 430
    приведение 430
    строки 427
    числовые 426
  для массивов 97
  преобразование
    между строкой и datetime 318
транспонирование массивов 108

У

укладывание 393
  определение 101, 393
  по другим осям 394
  установка элементов массива 397
унарные универсальные функции 111
унарные функции 110
универсальные функции 109, 398
  в pandas 150
  методы экземпляра 398
  пользовательские 400
универсальный режим новых строк 464
уровни
  группировка по 284
  определение 166
  сводная статистика 170
  сортировки 169
устойчивая сортировка 406

Ф

файловый ввод-вывод
  Web API 194
  базы данных 196
  в Python 463
  двоичные форматы данных 192
    HDF5 193
    Microsoft Excel 194
  массивов 119
    HDF5 412
    сохранение в двоичном формате 119
    сохранение и загрузка текстовых
      файлов 120
    файлы, спроецированные на память 410
  сохранение графиков в файле 256
  текстовые файлы 175
    HTML-файлы 188
    lxml, библиотека 188
    XML-файлы 190
    вывод данных 182
    данные в формате JSON 186
    формат с разделителями 184
    чтение порциями 181
факторная нагрузка 372
факторный анализ 372
факторы 372
фильтрация
  в pandas 143
  выбросов 224
  отсутствующих данных 163
финансовые приложения
  групповые преобразования 370
    квартильный анализ 373
    факторный анализ 372
  индексы доходности 368
  кумулятивная доходность 368
  линейная регрессия 380
  переформатирование данных 358
    выборка по состоянию на 364
    для нестандартных частот 361
    комбинирование данных 366
    с целью выравнивания 358
  роллинг фьючерсных контрактов 377
  скользящая корреляция 380
  стохастический граничный анализ 375
форма 383
функции 419, 452
  анонимные 456
  возврат нескольких значений 454
  замыкания 457

как объекты 455
каррирование 459
лямбда 456
область видимости 453
пространства имен 453
чтения в pandas 175

Х

хи-квадрат распределение 123
хронометраж программы 80
хэшируемость 448

Ч

частичное индексирование 167
частоты 326
неделя месяца 329
нестандартные 361
преобразование 336

Ш

шаговое представление 383

Э

экспоненциально взвешенные функции 353

Я

ядра 264

A

abs, функция 111
accumulate, метод 399
add_patch, метод 255
add_subplot, метод 246
add, метод 110, 148, 449
aggfunc, параметр 304
aggregate, метод 285, 287
all, метод 117, 399
alpha, аргумент 259
and, ключевое слово 429, 432
any, метод 117, 126, 225
append, метод 138, 439
apply, метод 51, 150, 161, 292, 296
apt, менеджер пакетов 22
arange, функция 96
arccosh, функция 111
arccos, функция 111
arcsinh, функция 111
arcsin, функция 111
arctanh, функция 111
arctan, функция 111
argmax, метод 116, 158
argmin, метод 116, 158
argsort, метод 153
arrow, функция 254
asarray, функция 97, 410
asfreq, метод 336, 347
astype, метод 98
average, способ 154
AxesSubplot, объект 247
axis, аргумент 211
axis, метод 156
ax, аргумент 259
a, режим открытия файла 464

B

Basemap, объект 271
bbox_inches, параметр 257
between_time, метод 365
bfill, метод 140
bisect, модуль 441
bottleneck, библиотека 352
break, ключевое слово 432
b, режим открытия файла 464

C

calendar, модуль 317
Categorical, объект 222
cat, команда Unix 177
cat, метод 236
ceil, функция 111
center, метод 237
Chaco, пакет 274
chunksize, аргумент 181
clock, функция 80
close, метод 245, 465
collections, модуль 447
cols, параметр 304
column_stack, функция 389
combinations, функция 463
combine_first, метод 199, 212
comment, аргумент 180
compile, метод 233
complex64, тип данных 98
complex128, тип данных 98

complex256, тип данных 98
concatenate, функция 388, 389
concat, функция 46, 199, 207, 208, 293
contains, метод 236
continue, ключевое слово 432
convention, аргумент 342
copysign, функция 112
copy, аргумент 204
copy, метод 134
corrwith, метод 159
corr, метод 159
cosh, функция 111
cos, функция 111
Counter, класс 32
count, метод 157, 231, 236, 286, 439
cov, метод 159
CPython 19
crosstab, функция 304
CSV-файлы 184, 268
cummax, метод 158
cummшт, метод 158
cumprod, метод 158
cumsum, метод 158
cut, функция 222, 224, 294, 311
Cython, проект 14, 414
c_, объект 389

D

DataFrame, структура данных 33, 38, 128, 131
  операции между DataFrame и Series 148
  слияние 200
dayfirst, аргумент 181
debug, функция 79
def, ключевое слово 452
delete, метод 139
del, ключевое слово 72, 134, 446
describe, метод 157, 269, 293
det, функция 122
diag, функция 122
difference, метод 449
diff, метод 138, 158
digitize, функция 408
divide, функция 111
div, метод 148
dot, функция 121, 122, 408
doublequote, параметр 186
dpi, параметр 257
dreload, функция 88
drop_duplicates, метод 217
dropna, аргумент 162
drop, метод 139, 142
dsplit, функция 389
dstack, функция 389
dumps, функция 187
duplicated, метод 217

E

edgecolor, параметр 257
eig, функция 122
empty, функция 97
encoding, аргумент 181
endswith, метод 231, 236
enumerate, функция 443
EPD (Enthought Python Distribution) 18
equal, функция 112
escapechar, параметр 186
ewma, функция 352
ewmcorr, функция 352
ewmcov, функция 352
ewmstd, функция 352
ewmvar, функция 352
ExcelFile, класс 194
except, блок 434
exec, ключевое слово 72
exit, команда 416
exp, функция 111
extend, метод 440
eye, функция 97

F

fabs, функция 111
facecolor, параметр 257
Factor, объект 295
ffill, метод 140
figsize, аргумент 260
Figure, объект 246, 249
fill_method, аргумент 342
fillna, аргумент 163
fillna, метод 33, 164, 220, 296, 346
fill_value, параметр 304
findall, метод 188, 232, 235, 236
finditer, метод 235
find, метод 230
first, способ 154
float16, тип данных 98
float32, тип данных 98

float64, тип данных 98
float128, тип данных 98
float, тип данных 97, 384, 426
float, функция 433
floor, функция 111
flush, метод 465
fmax, функция 112
fmin функция 112
fname, параметр 257
format, параметр 257
for, циклы 100, 115, 432, 451
from_csv, метод 184
frompyfunc, метод 400
functools, модуль 460

G

gcc, команда 23
getattr, функция 422
get_chunk, метод 182
get_dummies, метод 227, 229
get_value, метод 146
get_xlim, метод 251
get, метод 237, 447
greater_equal, функция 112
greater, функция 112
grid, аргумент 259
groupby, метод 51, 276, 324, 345, 373, 408, 462
  группировка по столбцу 281
  группировка с помощью функций 284
  обход групп 280
  передискретизация 345
  с помощью словарей 282

H

hasattr, функция 422
HDF5
  формат данных 193, 412
header, аргумент 180
hist, метод 264
how, аргумент 203, 342, 344
hsplit, функция 389
hstack, функция 389
HTML-блокнот в IPython 86

I

icol, метод 146, 172
idxmax, метод 158
idxmin, метод 158
ifilter, функция 463
if, предложение 431, 447
iget_value, метод 172
ignore_index, аргумент 211
imap, функция 463
import, директива
  в Python 422
  использование в этой книге 25
imshow, функция 113
in1d, метод 119
index_col, аргумент 180
index, метод 230, 231
insert метод 139, 439
insort метод 442
Int64Index, индексный объект 138
intersect1d, метод 119
intersection, метод 138, 449
int, тип данных 97, 426, 430
inv, функция 122
IIPython 17
  HTML-блокнот 86
  Qt-консоль 68
  выполнение команд оболочки 73
  завершение по нажатию клавиши Tab 59
  закладки на каталоги 75
  интеграция с matplotlib 68
  интеграция с редакторами и IDE 64
  интроспекция 60
  исполнение кода из буфера обмена 63
  история команд 70
  команда %run 61
  комбинации клавиш 64
  краткая справка 67
  магические команды 66
  обеспечение дружественность классов 90
  обратная трассировка 65
  перезагрузка зависимостей модуля 87
  профили 90
  советы по проектированию 88
  средства разработки 75
    отладчик 75
    построчное профилирование 83
    профилирование 82
    хронометраж 80
ipython_config.py, файл 91
irow, метод 146, 172

isdisjoint, метод 449
isfinite, функция 111
isinf, функция 111
isinstance, функция 421
isin, метод 139
is_monotonic, метод 139
isnan, функция 111
isnull, аргумент 163
isnull, функция 130
issubdtype, функция 384
issubset, метод 449
issuperset, метод 449
is_unique, метод 139
is, ключевое слово 424
iterator, аргумент 181
itertools, модуль 462
iter, функция 422
ix_, функция 108

J

join, метод 206, 230, 237
JSON (JavaScript Object Notation) 29, 187, 238

K

keep_date_col, аргумент 180
KeyboardInterrupt, исключение 62
kind, аргумент 259, 260, 342
kurt, метод 158

L

label, аргумент 259, 342, 343
last, метод 286
left_index, аргумент 204
left_on, аргумент 204
left, аргумент 203
len, метод 237, 284
less_equal, функция 112
less, функция 112
level, метод 156
level, параметр 284
limit, аргумент 342
linalg, модуль 121
line_profiler, расширение 84
list, функция 439
ljust, метод 231
loads, функция 29
load, метод 193
load, функция 119, 410
loffset, аргумент 342
log1p, функция 111
log2, функция 111
log10, функция 111
logical_and, функция 112
logical_not, функция 111
logical_or, функция 112
logical_xor, функция 112
logy, аргумент 259
log, функция 111
long, тип данных 426
lower, метод 231, 237
lstrip, метод 231, 237
lstsq, функция 122
lxml, библиотека 188

M

mad, метод 158
map, метод 151, 235, 456
margins, параметр 304
match, метод 233, 237
matplotlib 17, 245
  аннотирование 254
  интеграция с IPython 68
  конфигурирование 257
  метки осей 252
  названия осей 252
  подграфики 246
  пояснительные надписи 253
  риски 252
  сохранение в файле 256
  стили линий 249
matplotlibrc, файл 257
maximum, функция 110, 112
max, метод 116, 157, 286, 460
max, способ 154
mayavi, проект 274
mean, метод 115, 158, 278, 285, 286, 291
median, метод 158, 286
memmap, объект 410
mergesort', алгоритм сортировки 406, 407
meshgrid, функция 112
min, метод 116, 157, 286, 460
min, способ 154
modf, функция 111
mod, функция 112
MongoDB 198

MovieLens 1M, пример набора данных 38
mro, метод 384
MultiIndex, индексный объект 138
multiply, функция 111
mul, метод 148

N

names, аргумент 180, 211
NaN (не число) 116, 130, 162
na_values, аргумент 180
NA, тип данных 162
ncols, параметр 248
ndarray 94
  булево индексирование 104
  вырезание 100
  индексы 100
  операции между массивами 100
  перестановка осей 108
  прихотливое индексирование 107
  создание массивов 95
  типы данных 97
  транспонирование 108
None, тип данных 426, 430
NotebookCloud 86
notnull, аргумент 163
notnull, функция 130
npy, расширение имени файла 119
npz, расширение имени файла 119
nrows, параметр 181, 248
NumPy 16
  генерация случайных чисел 122
  линейная алгебра 121
  логические условия как операции с массивами 113
  массивы. *См.* массивы в NumPy
  массивы ndarray. *См.* ndarray
  методы булевых массивов 116
  обработка данных с применением массивов 112
  операции над матрицами 408
  производительность 412
    непрерывная память 412
    проект Cython 414
  случайное блуждание 123
  сообщества и конференции 24
  сортировка 403
  сортировка массивов 117
  статистические операции 115
  структурные массивы 401
  типы данных 384
  укладывание. *См.* укладывание
  универсальные функции 109, 398
    в pandas 150
    методы экземпляра 398
    пользовательские 400
  устранение дубликатов 118
  файловый ввод-вывод массивов 119

O

objectify, функция 188, 190
objs, аргумент 211
ols, функция 382
ones, функция 97
on, аргумент 204
open, функция 463
order, метод 406
or, ключевое слово 429, 432
outer, метод 399

P

pad, метод 237
pandas 16
  drop, метод 142
  reindex, функция 139
  арифметические операции и выравнивание данных 146
  выборка объектов 143
  доступ по целочисленному индексу 172
  иерархическое индексирование. *См.* иерархическое индексирование в pandas
  индексы 154
  обработка отсутствующих данных 162
    восполнение 164
    фильтрация 163
  построение графиков 258
    гистограммы 264
    графики плотности 264
    диаграммы рассеяния 266
    линейные графики 258
    столбчатые диаграммы 260
  применение универсальных функций NumPy 150
  ранжирование 151
  редукция 155
  сводные статистики

isin, метод 160
unique, метод 160
value_counts, метод 160
корреляция и ковариация 158
сортировка 151
структуры данных. *См.* структуры данных в pandas
фильтрация 143
Panel, структура данных 173
parse_dates, аргумент 180
parse, метод 318
partial, функция 460
pass, предложение 433
path, аргумент 180
pct_change, метод 158
pdb, отладчик 75
percentileofscore, функция 355
PeriodIndex, индексный объект 138, 339, 340
period_range, функция 335
Period, класс 335
permutations, функция 463
pickle, формат сериализации 192
pinv, функция 122
pivot_table, метод 40
plot, метод 34, 48, 54, 245, 258, 348
pop, метод 440, 446
pprint, модуль 90
prod, метод 286
put, функция 392
pydata, группа Google 24
pylab, режим 244
pymongo, драйвер 198
pyplot, модуль 245
pystatsmodels, список рассылки 24
Python
версии Python 2 и Python 3 23
генераторы 460
генераторные выражения 462
модуль itertools 462
достоинства 14
интегрированные среды разработки (IDE) 24
интерпретатор 416
кортежи 437
методы 439
распаковка 438
множества 448
множественное включение 450
необходимые библиотеки. *См.* библиотеки
поток управления. *См.* поток управления
семантика 417
атрибуты 421
динамическая типизация 422
директива импорта 422
изменяемые объекты 425
комментарии 418
методы 419
немедленное вычисление 424
объектная модель 418
операторы 423
отступы 417
переменные 419
ссылки 419
строго типизированный язык 421
функции 419
словари 445
словарное включение 450
списки. *См.* списки
списковое включение 450
типы данных 425
установка и настройка 18
Linux 22
Mac OS X 21
Windows 19
файловый ввод-вывод 463
функции. *См.* функции
функции последовательностей 443
enumerate 443
reversed 445
sorted 444
zip 444
pytz, библиотека 331

Q

qcut, метод 224, 294, 373
qr, функция 122
quantile, метод 158
quicksort', алгоритм сортировки 407
quotechar, параметр 186
quoting, параметр 186

R

randint, функция 123, 226
randn, функция 104, 123
rand, функция 123

range, функция 96, 435
ravel, метод 386
rc, метод 257
read_clipboard, функция 176
read_csv, функция 120, 175, 182, 287, 463
read_frame, функция 197
read_fwf, функция 176
readlines, метод 465
readshapefile, метод 272
read_table, функция 120, 175, 178, 184
read, метод 465
recfunctions, модуль 403
reduceat, метод 399
reduce, метод 398
regress, функция 301
reindex, метод 139, 146, 346, 361
reload, функция 88
remove, метод 440, 449
rename, метод 221
repeat, метод 237, 390
replace, метод 220, 231, 237
reset_index, метод 171
reshape, метод 385, 396
return, предложение 452
reversed, функция 445
re, модуль 232
rfind, метод 231
right_index, аргумент 204
right_on, аргумент 204
right, аргумент 203
rint, функция 111
rjust, метод 231
rollback, метод 330
rollforward, метод 330
rolling_apply, функция 352, 355
rolling_corr, функция 352, 380
rolling_count, функция 352
rolling_cov, функция 352
rolling_kurt, функция 352
rolling_mean, функция 352
rolling_median, функция 352
rolling_min, функция 352
rolling_quantile, функция 352
rolling_skew, функция 352
rolling_std, функция 352
rolling_sum, функция 352
rolling_var, функция 352
rot, аргумент 259
rows, параметр 304
rstrip, метод 231, 237
r_, объект 389
r, режим открытия файла 464
r+, режим открытия файла 464

S

savefig, метод 256
savez, функция 119
save, метод 192, 198
save, функция 119, 410
scatter_matrix, функция 267
scatter, метод 266
SciPy, библиотека 18
searchsorted, метод 407
search, метод 233, 235
seed, функция 123
seek, метод 465
Series, структура данных 128
  арифметические операции с DataFrame 148
  группировка с помощью 282
setattr, функция 422
setdefault, метод 447
setdiff1d, метод 119
set_index, метод 171, 216
set_title, метод 252
set_trace, функция 79
set_value, метод 146
set_xlabel, метод 252
set_xlim, метод 251
setxor1d, метод 119
set_xticklabels, метод 252
set_xticks, метод 252
set, функция 448
shape-файлы 272
sharex, параметр 248, 260
sharey, параметр 248, 260
shuffle, функция 123
sign, функция 111, 226
sinh, функция 111
sin, функция 111
size, метод 280
skew, метод 158
skipinitialspace, параметр 186
skipna, метод 156
skiprows, аргумент 180
slice, метод 237
solve, функция 122

sort_columns, аргумент 260
sorted, функция 444
sort_index, метод 151, 170, 406
sortlevel, функция 169
sort, аргумент 204
sort, метод 117, 403, 441, 457
split, метод 186, 230, 231, 235, 237, 388
split, функция 389
SQLite, база данных 196
sql, модуль 197
sqrt, функция 110, 111
square, функция 111
squeeze, аргумент 181
startswith, метод 231, 236
std, метод 116, 158, 286
strftime, метод 318
strip, метод 231, 237
strptime, метод 431
style, аргумент 259
subn, метод 235
subplot_kw, параметр 248
subplots_adjust, метод 249
subplots, метод 247
sub, метод 148, 235
suffixes, аргумент 204
sum, метод 115, 151, 156, 158, 285, 286, 359, 460
svd, функция 122
swapaxes, метод 109
swaplevel, метод 169

T

take, метод 226, 392
tanh, функция 111
tan, функция 111
tell, метод 465
text_content, метод 188
TextParser, класс 181, 182, 190
thousands, аргумент 181
thresh, аргумент 164
tile, функция 391
to_csv, метод 183
to_datetime, метод 319
to_panel, метод 174
to_period, метод 339
top, функция 293, 309
trace, функция 122
transform, метод 290
transpose, метод 108, 109
trellis, пакет 273
truncate, метод 323
try/except, блок 434
TypeError, исключение 99, 434
tz_convert, метод 333
tz_localize, метод 333

U

uint8, ти данных 98
uint16, ти данных 98
uint32, ти данных 98
uint64, ти данных 98
unicode, тип 30, 98, 426
uniform, функция 123
union, метод 119, 139, 228, 449
unique, метод 119, 139, 160, 306
unstack, метод 168
update, метод 367
upper, метод 231, 237
use_index, аргумент 259
U, режим открытия файла 464

V

ValueError, исключение 433
values, метод 446
values, параметр 303
var, метод 116, 158, 286
vectorize, функция 400
verbose, аргумент 181
verify_integrity, аргумент 211
vsplit, функция 389

W

where, функция 113, 211
writelines, метод 464, 465
writer, метод 186
write, метод 464, 465
w, режим открытия файла 464

X

Xcode 21
xlim, аргумент 259
xrange, функция 435
xs, метод 146
xticklabels, метод 251
xticks, аргумент 259

Y

yield, ключевое слово 461
ylim, аргумент 259
yticks, аргумент 259

Z

zeros, функция 97
zip, функция 444

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(499) 782-38-89**

Электронный адрес: **books@alians-kniga.ru**.

Уэс Маккинли

Python и анализ данных

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с английского *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 39,16.
Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф